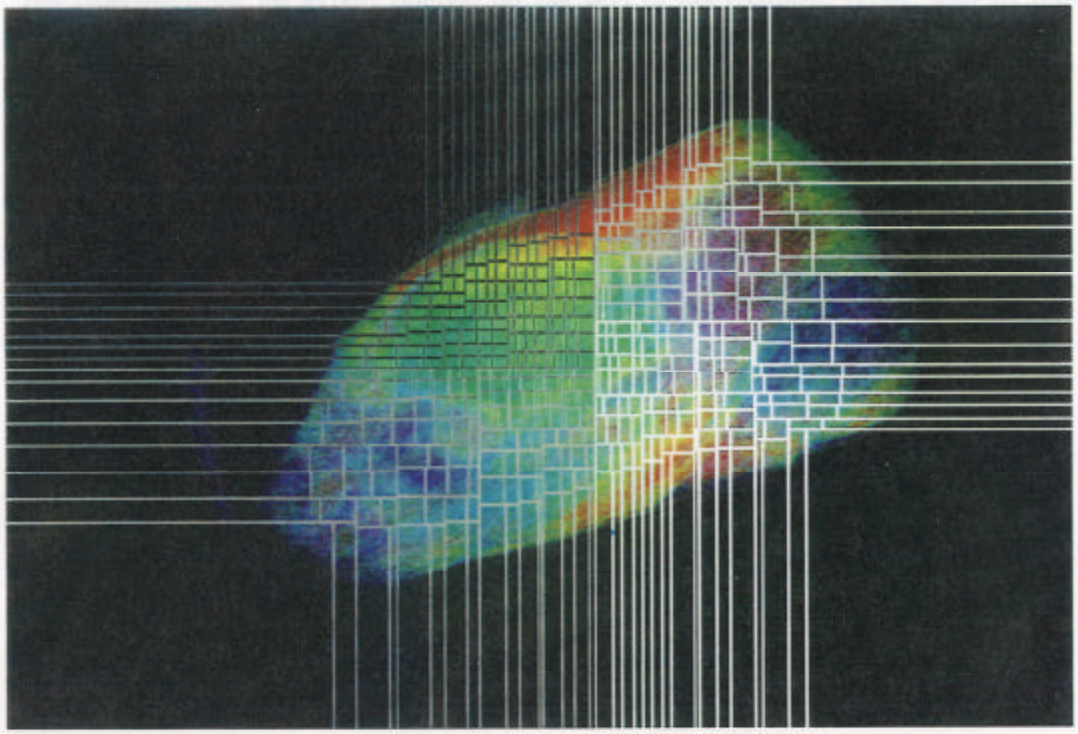
Immersing the Scientist in Data:
Interactive Visualization of Unstructured Scientific Data on Concurrent Architectures

Michael E. Palmer

Computer Science Department
California Institute of Technology
Caltech CS-TR-94-06



Head of Live Mouse (three-quarters profile view). This image is derived from Magnetic Resonance Imaging (MRI) data and contains over 1,000,000 tetrahedral voxels. The brain (green mass) and eyes (green spheres with blue centers) are easily identifiable. The image was rendered using 512 processors of the Intel Delta Machine; gray rectangles represent the areas of the screen rendered by each processor.

Acknowledgments

The author would like to thank Stephen Taylor for his deep involvement in this work; and Jeff Goldsmith for his expert opinion on the text of the complete thesis; also Al Barr, Dave Kirk, and Ulrich Neumann for their invaluable suggestions and advice. Thanks also to Russell Jacobs, John Allman, and Scott Fraser of the Biological Imaging Center, Beckman Institute, Caltech, for the MRI data sets.

Contents

Chapter 1

Introduction

1.1 Motivation

The complex three-dimensional computations now possible on modern supercomputers are swamping scientists with data — there is too much information to absorb and synthesize into understanding. Scanning huge piles of printed output is inadequate. Displaying two-dimensional cross sections of volume data is limited.

Both of these methods fail to exploit the sophisticated apparatus that has evolved in the human brain for gathering and processing visual information in order to understand complex three-dimensional, shadowed, colored, moving environments. A powerful and intuitive way to present complex, multidimensional scientific data would be to immerse the scientist in a *virtual environment.* The natural talent of the visual cortex for interpreting immersion in complex three-dimensional structures in real time could thereby be exploited.

A virtual environment is an interactive, computer-simulated illusion, that places the user in an artificially generated world. Examples of such environments might include: a flight over the surface of Mars, generated with radar data from actual space missions; a trip through the inside of a patient's heart before cardiac surgery, generated noninvasively from Magnetic Resonance Imaging (MRI) scans; or a seat on the wing of the Space Shuttle, watching its wing tip vortices, generated from Computational Fluid Dynamics (CFD) simulations.

Potential hardware components for such a system include those illustrated in Figure 1.1. The user may wear a helmet containing a display screen for each eye, and a glove or other pointing device. The position and orientation of the helmet and glove are tracked; and the computer displays to the helmet, in real time, a stereo view of the data appropriate to the current position and orientation of the helmet. The glove may be used to indicate a direction to move through the environment, or to manipulate objects in it.

1.2 Goals

This investigation is specifically concerned with a subproblem in the construction of a virtual environment for examining large scientific data sets: fast volume rendering of unstructured data. We seek good engineering compromises that allow these data sets to be clearly understood by scientists; we are not interested in rendering physical objects with a high degree of realism with respect to optical effects.

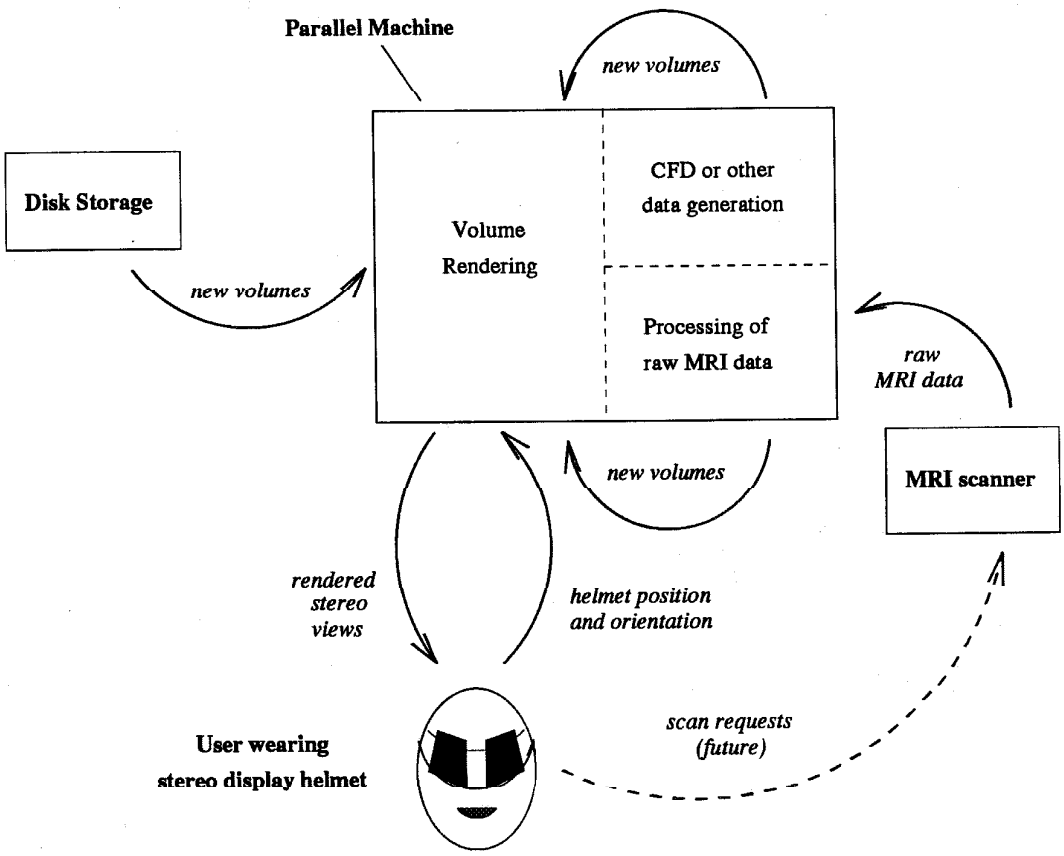


Figure 1.1: Immersion in a Virtual Environment

Concurrent algorithms deserve attention in this context because such algorithms are uniquely poised to exploit the parallelism inherent in volume rendering. Furthermore, many interesting scientific data sets require enormous storage space, and contemporary single-processor workstations commonly do not have the necessary memory or disk space. Finally, many scientists generate large data sets on concurrent computers, and therefore already have the machines available to do double duty as rendering engines.

There are three competing concerns in the design of an efficient concurrent algorithm: balancing workload among processors, balancing memory use among processors, and minimizing interprocessor communication. The central problem considered by this thesis is the design of rendering algorithms suitable for visualizing scientific data which address these concerns. To solve this problem involves finding a method to *partition* the rendering problem among many concurrent processors in way that minimizes interprocessor communication, yet allows for *load balancing* of workload workload and memory use.

1.3 Approach

The essence of the approach is to partition the computation of the rendering problem among concurrent processors in an unusual way, with an *image partition*, a partition that is fixed with respect to the screen image, rather than a *object partition*, one that is fixed with respect to the objects in the data set.

An object partition requires that rays being cast through the volume of data be passed from processor to processor. An image partition can be defined so that each ray travels entirely within a single processor, eliminating this communication. However, with each change of the observer's viewpoint, an image partition, which is fixed to the display screen, moves with respect to the data, requiring that some data be shifted to new processors. Whereas the amount of communication required by an object partition for each frame is relatively constant, the amount required by an image partition depends on the character of the change in the observer's viewpoint; the latter is therefore most efficient at rendering from the nearby viewpoints which are typically required by head mounted displays.

The interactive nature of foreseeable applications allowed acceptance of a certain constraint and the exploitation of a resulting form of coherence, which we call *viewpoint coherence*: that the position and orientation of a user wearing a head mounted display may be assumed not to move arbitrarily. From any given position and orientation, there is a finite range within which it can reasonably be supposed that the next position and orientation will be contained, simply because the rates at which the observer can move are limited.

1.4 Contributions

The main contribution of this thesis is a new approach to interactive, concurrent rendering of large, unstructured data sets, including the following: a method to partition the rendering problem across a concurrent computer; a method to redistribute the data as the viewpoint and the partitioning changes; and a method to dynamically load balance the rendering computation as it is running. The partitioning method uses a *recursive binary partition of image space* and is specifically designed to accommodate *dynamic load balancing*. Whereas the first of the two load balancing methods we implement uses a traditional scalar measure of workload distribution (i.e.,

a simple count of data voxels held by each processor), the second method we implement uses *center of mass vectors*; we find these to be superior to traditional scalar measures of workload distribution in accuracy; they may also be multiplied by an appropriate matrix to allow for *anticipation of data movement* to the next frame.

Chapter 2

Related Work

2.1 Volume Rendering Basics

Volume rendering is the process of producing a two-dimensional image of a three-dimensional scalar field or similar structure as it would be seen by an observer at a given position and orientation. This description implies two coexisting coordinate systems. The *object space coordinate system*, shown in Figure 2.1, is the coordinate system in which the data being viewed is defined.

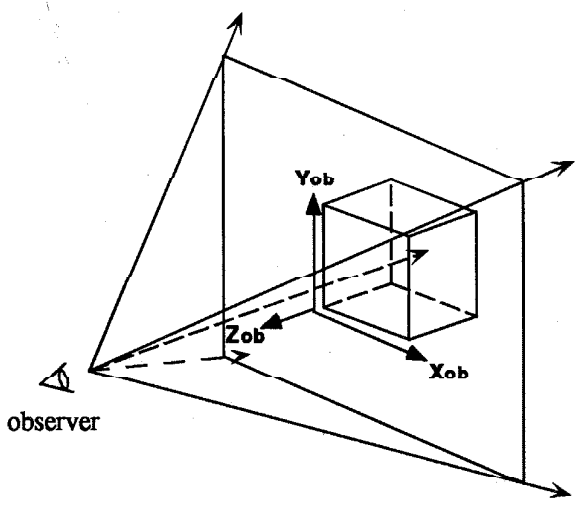


Figure 2.1: Object Space

In this figure, the data set consists of a single cube. For convenience, the axes of the object space coordinate system are aligned with the edges of the cube. We will refer to the position and orientation of the observer collectively as the *viewpoint*. The viewpoint will move in object space as the data set is observed from different locations and orientations. If a perspective projection is used, as in the figure, rays projected out from the eye of the viewer will diverge in object

space.

The *image space coordinate system*, illustrated in Figure 2.2, is fixed to the viewpoint of the observer. The origin of the image space coordinate system is the location of the observer, the Z direction is defined as the "forward" direction of the viewer, and the Y direction is define as the "up" direction of the viewer. Note that the perspective projection used in the figure makes the back face of the cube appear smaller than the front face. Rays projected from the eye are parallel in image space. Given the viewpoint of the observer (i.e., the observer's position and orientation), and an additional scalar specifying the magnitude of the perspective distortion, it is possible to compute a matrix which maps vectors in object space to vectors in image space; this matrix is called the *view matrix*.

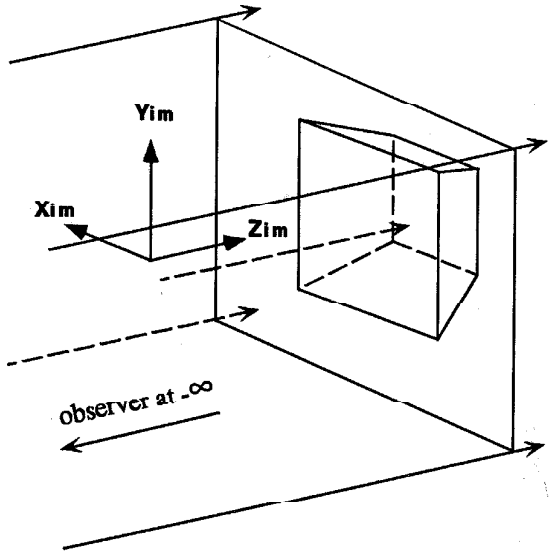


Figure 2.2: Image Space

2.1.1 Techniques of Volume Rendering

Early computer graphics work grappled with solving the *hidden line* problem: a computer making a line-drawing to render a set of polyhedral objects from a given viewpoint must be able to decide which lines or parts of lines will be obscured, and should not be drawn. Related, but more difficult, is the *hidden surface* problem: how to decide which polygons or parts of polygons should not be drawn when constructing an image out of polygons. A classic paper which categorizes a variety of early methods to attack these problems was written by Sutherland et al. [Sutherland74], who catalog several forms of coherence in data sets that can be exploited to solve the hidden-line and hidden-surface problems efficiently. Several of the forms of coherence described in that paper (here slightly modified) are:

- Object coherence
  That objects tend to be connected, bounded bodies.

- Area coherence
  That the two-dimensional projection of a three-dimensional body is a connected, bounded figure.

- Scan line coherence
  That adjacent rays are likely to intersect the same objects.

- Frame coherence
  That one frame of an animated sequence is likely to resemble the previous frame.

A more difficult problem that was not yet much explored when that paper was written could be called *hidden volume-density* problem — how to determine which volumes of a three-dimensional data set contribute to a pixel.

Modelling 3D Scalar Fields for Volume Rendering

Many scientific data sets consist of a collection of scalar elements, each associated with a discrete region of space. This discrete representation may approximate a continuous scalar field that exists in the physical world.

There are two common ways in which scalar fields are modelled in volume rendering. First is the extraction of sets of surfaces on which the scalar field assumes some constant value. This process is called an *isosurface extraction* [Lorensen87]. This technique can distill desired features from a large amount of data, but the extraction process may introduce artifacts. Moreover, a holistic view of the data is sometimes desired. This thesis focusses on an alternative called *direct volume rendering* [Upson88, Sabella88, Drebin88, Levoy88, Max90, Wilhelms91a]. This technique treats the scalar field as a cloud of translucent material of varying opacity and color. A *color map* and *opacity map* map a scalar volume into a *RGBα volume*, where each point has an associated RGB value, or color, and α value, or opacity. By adjusting the color and opacity mappings, different features of the data may be emphasized in the context of the whole.

Object-Order Composition

A class of methods called *object-order composition* are most often used for volume rendering [Upson88, Drebin88, Max90, Westover90, Wilhelms91a, Laur91], and are illustrated in Figure 2.3. These methods sort the objects in the data set such that any object comes before those which it obscures; objects are then be taken off the list and their projected image composed directly onto the array of display pixels. Complications arise, however, if the objects are not convex, because it is then possible to have two objects A and B such that A and B both partially obscure each other, so that they cannot be sorted onto such a list. (A solution is to subdivide such objects.) This thesis work concentrates on object-order composition.

Image-Order Composition

Another class of methods called *image-order composition* may also used for volume rendering [Sabella88, Levoy88]. These methods calculate the color of each pixel by following a ray

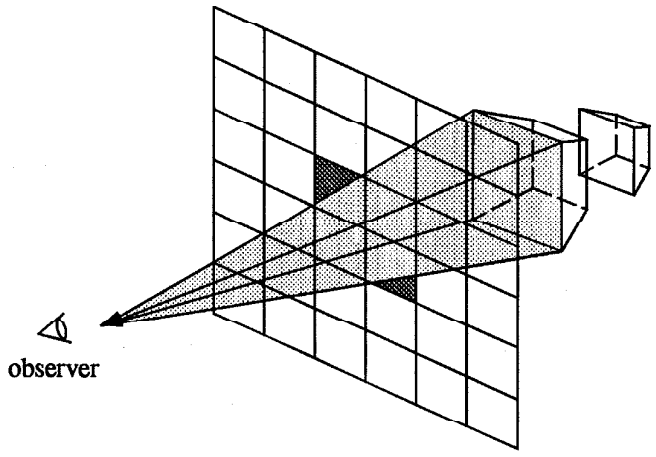


Figure 2.3: Object-Order Composition

through the pixel and composing the effects of the objects in the order that the ray intersects them. (It is assumed here, for simplicity, that a single ray is cast per pixel.) This method of composition is illustrated in Figure 2.4. As the ray progresses through the data, the database must be repeatedly searched to identify the next object which the ray will hit.

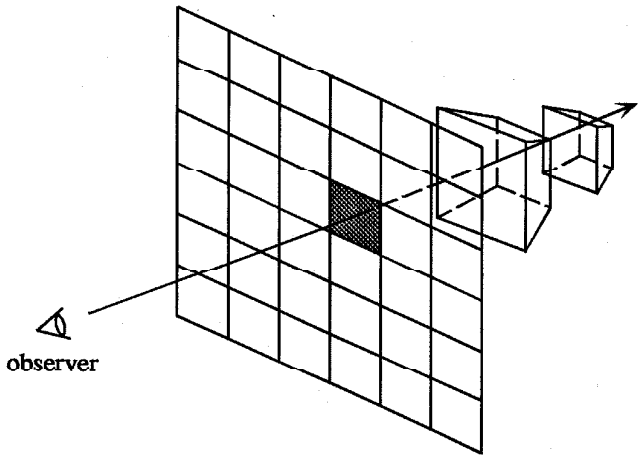


Figure 2.4: Image-Order Composition

Ray Tracing

Although this thesis is not directly concerned with *ray tracing*, we describe it briefly here, because we will later mention it as the historical context for the development of certain methods to partition the rendering computation. The basic algorithm was first described by Appel [Appel68] and later improved [Bouknight70, Kay79, Whitted80, Rubin80].

Whereas, in the physical world, light rays leave their sources, interact with objects, and finally enter the eye of the observer, ray tracing algorithms follow the rays in the opposite direction: starting at the eye of the viewer, passing through pixels of the display screen, interacting with the objects, and ending at the light sources, as in Figure 2.5. The reason for this reversal is to avoid the expense of tracing the many rays which leave the light sources and never reach the eye of the viewer. Modeling of physical phenomena like the reflection, refraction, and transmission of light from and through various materials can produce highly realistic images. Rays accumulate color and opacity as they pass through translucent objects. Rays may reflect and refract off of objects, spawning more rays recursively. The realistic rendering of natural clouds and haze was first described in the context of ray tracing [Blinn82, Kajiya84, Max86]. These methods were later simplified and adapted to rendering of scientific scalar data.

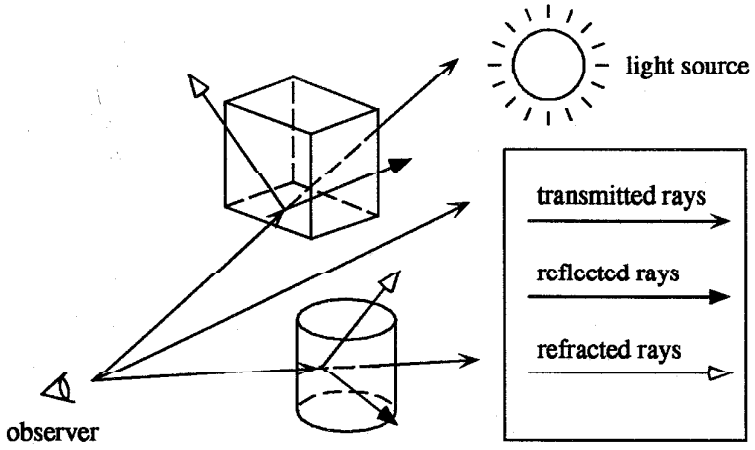


Figure 2.5: Paths of Rays in Ray Tracing

2.1.2 Representations of Scientific Data

The discrete elements in a three-dimensional scientific data set are called *voxels* if their values are constant within their boundaries, or *cells* if their value is assumed to vary predictably (e.g. linearly) within their boundaries.

Wilhelms [Wilhelms91b] classifies the grids on which volumetric data is commonly defined into four types: *regular*, *rectilinear*, *curvilinear*, and *unstructured* grids. A regular grid consists of identical rectangular solids. Rectilinear grids are more general, and may be generated by warping the distances along the axes of a regular grid (under certain constraints, e.g. invertibility of the

warping). Curvilinear grids may be generated by a (constrained) warping in space of a regular grid. For all of these first three types of grid, connectivity between neighboring voxels is explicit, and the voxels are ordered in each dimension. For unstructured grids, connectivity is not explicit. These grids are arbitrary polyhedral decompositions of space which, Williams [Williams90] notes, may not be depth-orderable.

Most important for volume rendering algorithms is the presence or absence of information about a voxel's neighbors. In image-order composition, when a ray exits an object, the algorithm must find the next object that the ray intersects; in object-order composition, the algorithm must sort the voxels in the order that they obscure one another. On regular and rectilinear grids, traversal of voxels, point-location, and interpolation are straightforward, because the connectivity is simple and regular. Therefore, both types of volume rendering algorithms can be implemented efficiently. Curvilinear grids present more complication for traversal and interpolation, but a voxel's neighbors are still explicitly defined. Data defined on unstructured grids, however, is often presented as a one-dimensional list of voxels. The voxels are not implicitly ordered and may have arbitrary numbers of neighbors at arbitrary position. A voxel may be isolated in space and have no neighbors that touch it. Furthermore, there may be large empty spaces inside the data set. Since image-order rendering algorithms would have to sort an unstructured data set for each ray, object-order rendering is more commonly used for these data sets. This thesis work concentrates on the volume rendering of unstructured data.

Shading and Reflection in Scientific Visualization

Complex shading and reflection models, introduced to the field of ray tracing in order to provide realistic rendering of certain physical phenomena, may not be appropriate to the visualization of scientific data [Sabella88]. Certainly, shading is a valuable tool to draw out the features of a contoured surface that would otherwise appear blank. However, it is debatable whether, for instance, multiple light sources are helpful rather than simply confusing. Reflectivity may have no natural meaning in the context of the data; it could be given a meaning, just as color in "false color" images is given meaning, but reflectivity is a particularly difficult characteristic for the eye to interpret quantitatively. Reflections have the undesirable characteristic of causing repetition of patterns in the data far from their point of origin — placing apparent meaning where it does not exist, detracting from the overall comprehension of the data.

In general, the simpler the rendering algorithm, the easier it will be for the scientist to reason whether an apparent pattern reflects the character of the data, or is simply an artifact of rendering. This pressure toward algorithmic simplicity is counterbalanced by the need for sufficient complexity to completely and unambiguously represent the data.

2.2 Concurrent Volume Rendering

2.2.1 Special-Purpose versus General-Purpose Hardware

Concurrent algorithms are uniquely poised to exploit the parallelism inherent in volume rendering. Methods that do not require special-purpose concurrent hardware are of particular interest: Many users with large scientific data sets originally generated them on general-purpose multicomputers — these computers could cost-effectively also serve as visualization engines. Furthermore, Green and Paddon [Green90] point out that, judging from the recent proliferation of

literature on new optimization methods, new lighting models, and more complex environments for rendering, the potential for further development is far from exhausted. Special-purpose rendering machines risk being unable to take advantage of algorithmic developments.

A tremendous variety of concurrent rendering algorithms have been proposed, using special-purpose and general-purpose concurrent hardware. Only in the past several years, however, has actual implementation on concurrent hardware, rather than software simulation, become the norm. Two early exceptions which were implemented in hardware are Nishimura et al. [Nishimura83] on special-purpose hardware, and Salmon and Goldsmith [Salmon88], on general-purpose hardware.

2.2.2 Concerns in the Design of Concurrent Algorithms

There are three major concerns in the design of an efficient concurrent algorithm:

- The workload of computation must be evenly balanced among the processors.

  If one processor has more work to do than the others, then most processors will be left idle waiting for it to finish. Ideal load balance may be defined by the condition that the processor with the most work has the same amount of work as the average processor; this implies that all processors have an equal amount.

- Memory use must be evenly balanced among processors in order to make full use of the storage capabilities of the machine.

  This is of particular concern for large scientific data sets, which chronically push the limits of storage.

- Interprocessor communication must be minimized and balanced among processors.

  On certain concurrent architectures, interprocessor communication is at least two orders of magnitude as expensive as a local memory fetch. Communication is often the bottleneck in algorithms that would at first glance seem efficient.

2.2.3 Partitioning the Rendering Problem among Concurrent Processors

Data Redundancy

Perhaps the simplest way to parallelize volume rendering is to take a serial implementation written for a particular workstation and run it concurrently on a lab full of workstations, assigning each workstation a subsequence of the frames of a complete movie. This is not an interactive solution, however, since the frames will have to be edited together later.

The simplest method using dedicated parallel hardware is to replicate the entire database of objects on every processor, and assign each processor some subset of rays or objects to compose. Once the data has been distributed in this redundant way, composition requires no communication between processors. However, for a collection of N processors, this is a wasteful N-fold replication of the data in the collective memory of the processors. Multicomputers consisting of hundreds or thousands of processors typically have limited memory per processor, and large databases cannot be rendered by such methods.

Image and Object Partitions

To avoid such redundancy, the data set may be partitioned among the parallel processors. Partitioning methods for rendering were devised independently of concurrent algorithms, in the context of ray tracing, as a method to reduce the number of ray-object intersection tests. Partitions can be classified into those which partition the objects with bounding volume hierarchies [Rubin80, Kay86], and those which partition space itself through spatial subdivision [Glassner84, Kaplan85, Fujimoto86]. If it can be determined that a ray does not intersect any object in a given part of the hierarchy, or in a given part of space, then no such objects need to be explicitly tested, and the overall number of tests can be greatly reduced.

One approach taken by concurrent rendering algorithms has been to assign a fixed portion of image space to each processor; this is called an *image partition* [Levoy89, Nieh92, Neumann93]. In the image partition used in this thesis, each processor is assigned a contiguous rectangular area of the display screen and owns the volume of image space directly behind it. Each processor receives a copy of all voxels that fall within its volume of image space. For such a partition, neither image-order nor object-order composition methods require communication to perform the composition part of the algorithm, once each processor has all the voxels in its area of image space. However, communication needs do arise in passing all the appropriate data to each processor, because the relationship between image space and object space, and therefore the data required by each processor, changes with each shift in viewpoint.

Another approach is to assign the objects in the data set to fixed processors. Such a partitioning of the problem is called an *object partition.* Concurrent algorithms have been described which use both types of object partition: Bounding volume hierarchies were used in [Goldsmith87] and [Scherson88]; whereas spatial subdivision was used in [Dippé84, Nemoto86, Cleary86, Kobayashi87, Priol89] and others. Image-order composition is more commonly used for these partitions than object-order composition; in image-order composition, rays will, in general, intersect data held by many different processors. Therefore, rays cast through the data must be communicated between processors in order to be tested against all the relevant data elements.

For an image partition, the amount of communication required per frame depends on how the viewpoint changes, whereas for an object partition, a relatively constant amount of communication is required, independent of the change in viewpoint.

Methods of spatial subdivision can be further divided into those which employ a uniform, static subdivision [Cleary86, Kobayashi88], and those that divide space in an adaptive manner, either by iterative [Nemoto86, Dippé84, Nieh92], or recursive [Kobayashi87, Priol89] subdivision methods. Uniform subdivision is naturally static and makes no provision for balancing load among processors; however, traversal of processors is efficient. Adaptive subdivisions may be adapted to the data just once at initialization [Priol89], or dynamically in response to changing conditions of workload during execution [Dippé84, Nemoto86, Nieh92]. Calculating the proper adaptation incurs cost, and adaptive subdivisions are more expensive to traverse, but when the data is non-uniformly distributed in space, adaptive subdivisions can improve the load balance. It is likely that neither type of subdivision is best in all cases, but that uniform subdivision is more efficient when the data is uniformly distributed, and that adaptive subdivision is more efficient otherwise. Likewise, statically balanced adaptive algorithms probably win when load is relatively unvarying, whereas dynamically balanced ones probably recoup their extra costs through better load balance when load varies dramatically during execution. This thesis work

implements dynamically adaptive load balancing to meet the demands of changing workload common to image partitions and unstructured concurrent computing in general.

Chapter 3

New Concurrent Methods for Volume Rendering

3.1 Goals and Constraints

The primary goal of this research is the development of concurrent algorithms to aid the understanding and interpretation of scientific data. Of particular interest are concurrent methods that could be used for the simulation of an interactive Virtual Environment.

The focus is on the visualization of unstructured data because of the interesting challenges of unstructured concurrent computing in general. One such challenge is the dynamic balancing of the computational and memory loads of unstructured computations as they are running. The type of data has been limited to scalar elements as opposed to vectors, as a scalar field can be intuitively represented as a cloud of varying density and color. This direct volume rendering was chosen rather than isosurface extraction because it gives a holistic view of the the data set; it is also better suited to the visualization of unstructured data.

Concurrent algorithms require several special algorithmic considerations: computational workload must be balanced among all processors to fully exploit the computational resources of the machine; memory use must be similarly balanced to exploit the available storage capacity; finally, interprocessor communication must be minimized and balanced among processors.

3.2 Overview of the Algorithm

3.2.1 Using an Image Partition versus an Object Partition

In view of these goals and constraints, this thesis work takes a different approach to concurrent volume rendering than the vast majority of other work. The principal difference is the method of partitioning the data and rendering work across the concurrent machine: while most other algorithms have used an object partition — one which is fixed with respect to the objects in the data base, this work uses an image partition — one which is fixed with respect to the screen image.

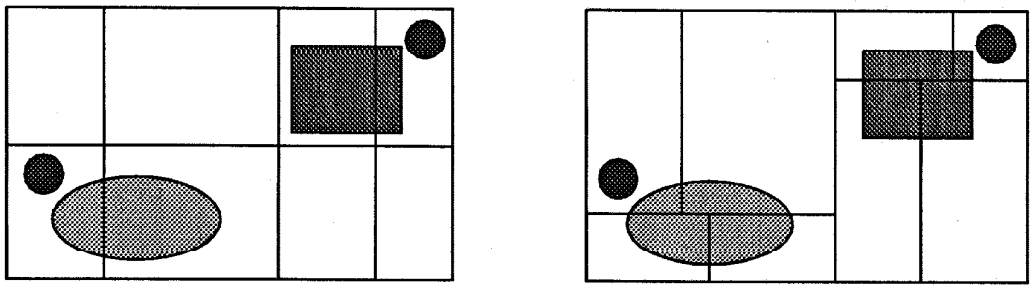
One reason why the object partition has been so popular is that the image partition does not allow for the reflection and refraction effects desired by some algorithm designers. However, another reason is apparently that it was assumed that moving the data at each change in

viewpoint would be more expensive than moving the rays. When single still images are being generated, this is very likely true. This is not necessarily the case, however, if an animated sequence is being generated and one assumes another form of coherence, which we call *viewpoint coherence*, the assumption that *the viewpoint does not change greatly from frame to frame.* If the change in view is vanishingly small between frames, then a vanishingly small amount of data movement would be required between frames. If an object partition were used, on the other hand, the same number of rays would always have to be moved no matter how slight the change in viewpoint.

Assuming a maximum head-turning rate of 300 degrees/second and a frame rate of 30Hz, the maximum amount of head rotation from frame to frame will be 10 degrees/frame. When the observer is focusing on an localized area of interest, something on the order of 2 or 3 degrees/frame might be expected. How this translates into percentage of data moved, or into a number of physical messages, will depend on the partitioning, the method of transporting the data, the character of the viewpoint change, and number of data elements in view. As one looks forward to volume rendering at real-time rates in the future, the constraint of viewpoint coherence becomes more acceptable.

3.2.2 Recursive Subdivision of Image Space

The diagram on the left of Figure 3.1 illustrates an inflexible way to divide the screen. The darkened areas represent objects, which require a concentration of work load. This inflexible subdivision method divides the screen with lines the run the full length of the screen (left to right) and the full height of the screen (top to bottom). Assuming a two-dimensional division, no matter how the lines are moved, the inflexible method cannot assign all processors an equal amount of work for this distribution of objects — some processors will always have much more than the average. The diagram on the right of the figure shows the well-balanced positioning of lines that can be achieved by the flexible subdivision method presented in this thesis. It can approximate an arbitrary distribution of processor power much more closely; such flexibility is required to accommodate load balancing.



an inflexible partitioning method

a flexible partitioning method

Figure 3.1: Inflexible and Flexible Methods to Subdivide the Screen

The flexible partitioning method we used works in the following way: The screen is cut

recursively into two parts, by a series of $\log N$ cuts, where N is the number of processors. Each processor is assigned the volume of image space behind one rectangle. The first, and subsequent odd divisions, are parallel to the Y axis; the second, and subsequent even divisions, are parallel to the X axis. This is similar to the binary space partition of [Kaplan85].

Figure 3.2 illustrates the ownership of areas of the screen and of volumes of image space. Here there are $N = 8$ processors, numbered from zero to seven. The left of the figure shows assignments of screen areas to processors, while the right shows the corresponding assignments of volumes of image space to processors. A processor owns the entire volume of image space behind its portion of screen area. In the figure, the entire screen is recursively divided by $\log N = 3$ sets of lines: first by the line A1, then by the lines B1 and B2, and finally by the lines C1, C2, C3 and C4.

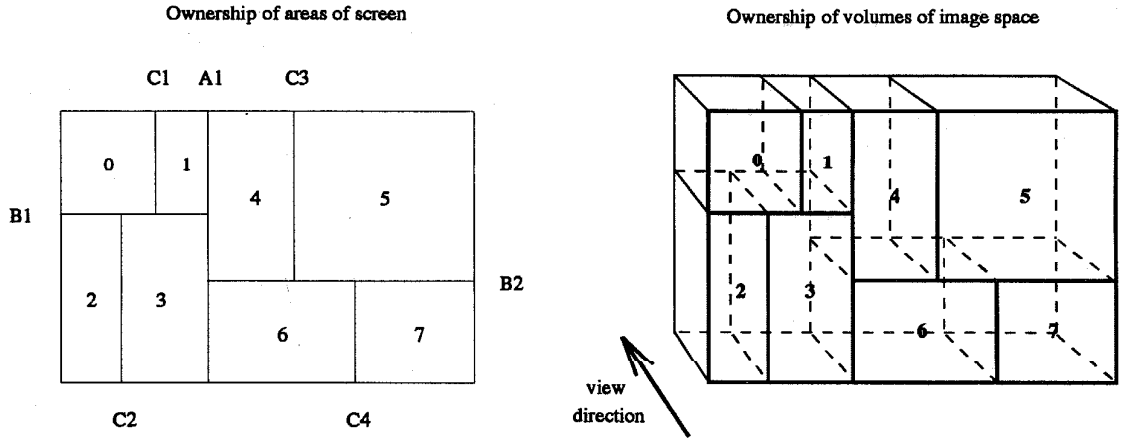


Figure 3.2: Subdivision of Screen Area and Image Space

The recursive subdivision is primarily designed to accommodate dynamic load balancing. In response to changes in load, as detailed below, the lines dividing the screen will be shifted and ownerships of volume will change correspondingly. Our subdivision of the screen allows flexible distribution of processor power to concentrations of work load.

3.2.3 Overview of Main Algorithmic Steps

The basic cycle repeated to generate an image of a data set from a particular viewpoint is the following. It is assumed that a data set of unstructured voxels has already been loaded onto the machine.

- *Broadcast current view matrix to all processors*
  The view matrix is the transformation between object space and the current image space and represents the viewer's position, view direction, and up direction.
- *Calculate new positions of voxels in image space*
  Each of the voxels' vertices, which are stored in object space coordinates, is multiplied by

the current view matrix, to find the vertex' position in image space.

- *Transport voxels to correct owners*
  Each processor owns a subvolume of image space. When the mapping between object space and image space changes, some voxels may move into different subvolumes of image space; here they are sent to their new owners.

- *Compose Voxels*
  When a processor has all the voxels that fall into its subvolume of image space, it may render its portion of the display. Our implementation uses object-order composition.

- *Deliver each processor's portion of screen image to frame buffer hardware*

- *Load Balance*
  Based on the amount of work needed to compose the current frame, the lines subdividing areas of the screen are moved so that the workload is evenly balanced among the processors.

- *Recycle voxels or load new volume*
  If the data set is an element of a time sequence of data, the next data set in the sequence is loaded. Otherwise, the same voxels are recycled to be rendered from the next viewpoint.

3.2.4 A System of Message Queues

The above algorithm is implemented as a system of message queues. Around this system of queues travel voxels, which each contain the following information:

- A scalar value valid for the volume within that voxel
- The vertices in object space coordinates
- The vertices in current image space coordinates
- The center in object space coordinates
- The center in current image space coordinates
- Voxel type (Master, Shadow, or Sleeping)

Figure 3.3 shows the algorithm executed by each processor, organized in terms of message queues. Each revolution through the main cycle in the figure represents the rendering of the data from one viewpoint.

When voxels are first loaded from disk or from another source, they move onto the *load-voxel-queue*, which serves as temporary storage until all voxels have been loaded; while one set of voxels is being loaded, a previous set can continue to flow around the cycle of queues, as views of it continue to be rendered. When voxels are first loaded, there is no current image space defined for them, and the information about their centers and vertices in image space coordinates is empty.

At the receipt of the first new view request after a new data set has been completely loaded, then the old data set (if there is one) is thrown out and the new one is copied from the *load-voxel-queue* to the *trans-voxel-queue*; this may be done with a single pointer copy. The view request

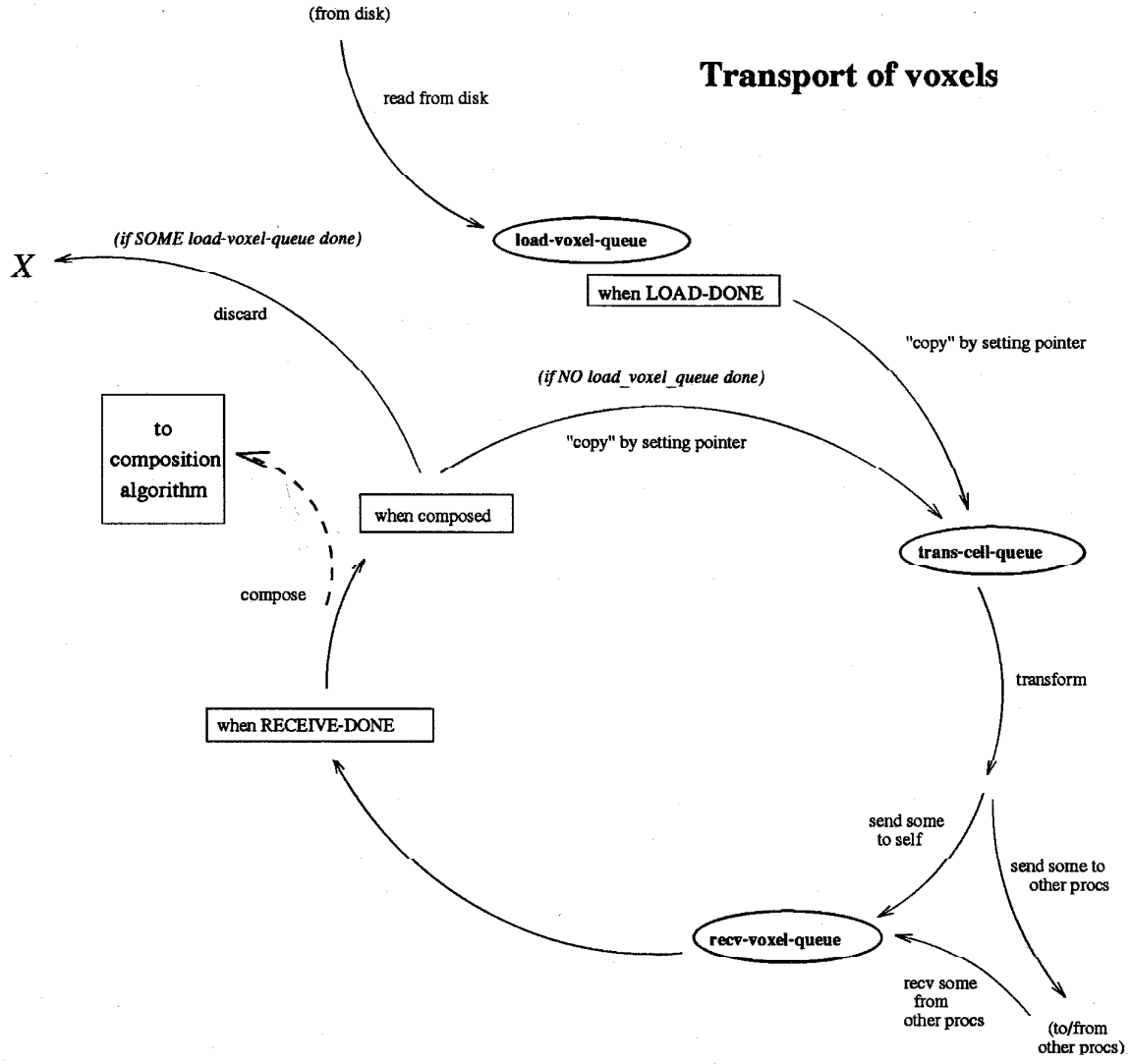


Figure 3.3: Voxels Pass through a System of Message Queues

contains the transformation matrix mapping between object space and the current image space. As each voxel on the *trans-voxel-queue* is processed, the coordinates of its center and vertices in the current image space are calculated. If a voxel is found to fall outside the current boundaries of the screen, then it will not be visible in the current image; it is labeled *sleeping* and sent to a randomly chosen processor for storage — this will on tend to place an equal number of sleeping voxels on all processors, balancing memory use.

A voxel's *owner* is the processor within whose subvolume of image space the voxel falls. All voxels which are not sleeping are sent to their proper owner. If a voxel intersects more than one of the subvolumes of image space, then one *master* and one or more *shadow* copies are generated and sent to their appropriate owners. The owners receive the voxels and store them on the *recv-voxel-queue*.

All processors wait for a synchronization signal that indicates that all voxels have been received by their owners; composition may then begin. The current implementation used object-order composition, but the methods introduced here apply equally well to image-order composition.

When each processor finishes composition, it sends the area of the screen image it has generated to the frame buffer for display. When the next view request is received by processor 0, it decides for all processors whether or not to *recycle* the voxels in the *recv-voxel-queue*; the voxels are recycled if no new data set has been loaded. If the voxels in the *recv-voxel-queue* are to be recycled, then they are copied to the *trans-voxel-queue*, and the cycle begins again.

3.3 Details of Algorithm

The following section gives greater detail on those portions of the algorithm which are novel contributions of this thesis work.

3.3.1 Transport of Voxels through Hopping

When the observer's viewpoint changes, the mapping between object space and image space changes. All voxels are given a new position in image space based on this new mapping and their fixed object space coordinates. Some voxels will have traversed one or more of the planes which subdivide image space. This will require that the processor holding such a voxel pass that voxel to its new owner.

In the interest of saving memory, simplifying load balancing, and simplifying the search for a voxel's correct owner(s), the division of the screen is stored as a recursive hierarchy of line positions rather than as a list of rectangles. Rather than having each processor maintain and search a list of N rectangles representing all the subvolumes of image space (where N is the number of processor), we have each processor store only the positions of $\log N$ of the lines subdividing the screen. These lines were previously illustrated in figure 3.2. The lines that are stored by each processor, for $N = 8$, are shown to the right of Figure 3.4. Note that no processor has information about the entire subdivision of the screen.

When a processor discovers that it holds a voxel which it no longer owns, or for which shadow copies will need to be generated, it sends the voxel to the proper owner(s) by a series of $\log N$ hops; the path of these hops is shown on the left of Figure 3.4. A processor that needs to send a voxel to a new owner compares that voxel's X position with line A1, as shown to the left of the figure. (Note on the right of the figure, that all processors know the location of line A1.) If that

voxel falls to the left of line A1, the processor passes it to one of processors 0 through 3, which compare its Y position to line B1. (Note on the right of the figure that processors 0 through 3 all know the location of line B1.) If the voxel instead falls to the right of line A1, the processor passes it to one of processors 4 through 7, which compare its Y position to line B2. (Note that processors 4 through 7 all know the location of line B2.) After $\log N$ hops, the voxel reaches its owner at the bottom of the tree.

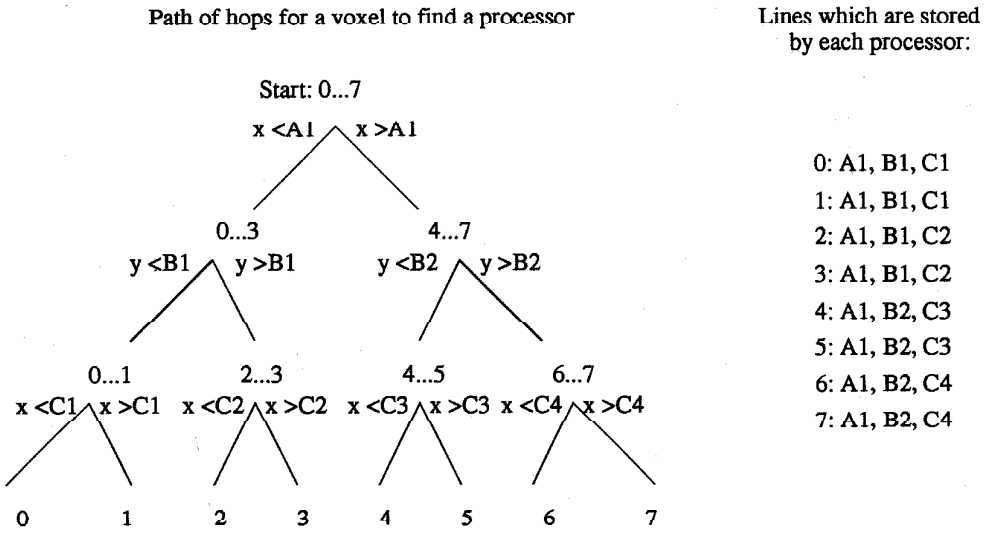


Figure 3.4: The Hop Tree, and the Line Positions Stored by Each Processor

3.3.2 Master and Shadow Copies of Voxels

If a voxel intersects one of the lines dividing the screen, then it will fall into the subvolumes of more than one processor. The processor that detects this condition for a given voxel and a given line will generate a second copy of the voxel. One copy will be labeled the *master* and sent to the child on one side of the line; the other copy will be labeled the *shadow* and sent to the other child. Both copies of the voxel continue hopping down the tree until they reach their final destinations. If a voxel already marked as a shadow must be duplicated again, then both copies are labeled shadow, in order to maintain only one master copy of each voxel. When the viewpoint changes again, the shadow copies are discarded and only the master copy is reused — thus voxels are prevented from multiplying uncontrollably. Master and shadow copies of a voxel are illustrated in Figure 3.5.

3.3.3 Load Balancing

Any efficient concurrent algorithm requires attention to balancing the computational workload among the processors. In our method, this involves two distinct phases: communication of

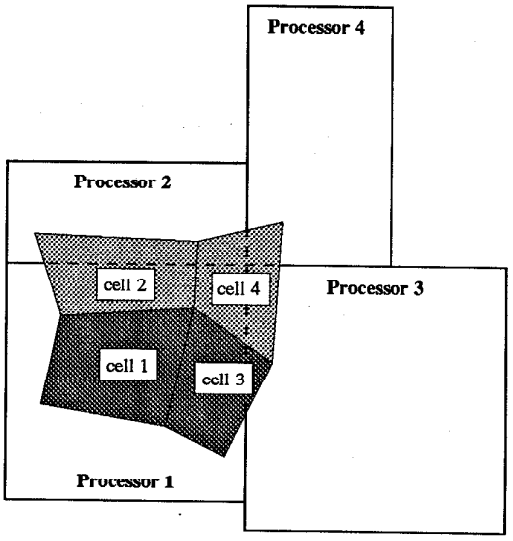


Figure 3.5: Master and Shadow Copies of Voxels

workload distribution information up the tree, and balancing of lines dividing the screen, back down the tree.

Communication of work up tree

We achieve load balancing by the movement of the lines dividing areas of the display screen, such as lines A1, B1, B2, C1, C2, C3, and C4 in Figure 3.2. A single processor is responsible for deciding the position of each line. Simple algebraic formulas determine the "parent" of each line and its two "children". To the left of Figure 3.4, for example, processor 0 is the first processor in the sequence 0...7, and is therefore the parent responsible for deciding the proper placement of line A1; its two children at level A are processor 0 itself (the first processor in the sequence 0...3), and processor 4 (the first processor in the sequence 4...7). These two processors are responsible placing for lines B1 and B2, respectively; and so on for the four C lines, whose parents are processors 0, 2, 4, and 6, respectively.

Reports of workload travel up the tree by special messages. At the leaves of the tree to the left of Figure 3.4, a process' local load is determined by some measure of the expense it took to render the most recent frame. In the initial implementation, a simple count of voxels held by each processor was used, since this is roughly proportional to computational workload when using object-order composition. All eight processors pass this load up to their parent at level C. The parents at level C (processors 0, 2, 4, and 6) sum the loads of their children to compute their loads at level B. These are passed to their parents at level B (processors 0 and 4). Processors 0 and 4 sum these to compute their loads at level A, which are passed up to processor 0, the parent at level A.

Balancing of lines down the tree

Updated line positions travel back down the tree. When the observer requests a view, this initiates a cascade of messages down the load balancing tree. These messages include the transformation matrix between object space and image space, and the new line position information, which is updated as it passes through the parents at different levels down the tree. A parent at any level will move its line to a position appropriate to the workloads of its two children. Where parents place the lines depends on the local load information they have and where the lines at higher levels have already been moved. When the cascade of messages reaches the leaf processors at the bottom, the new line position information is complete and personalized for each process: for example, for $N = 8$, each processor will receive the position of the three lines shown to the right of Figure 3.4. Each process stores a personalized subset of the entire tree.

Note that there is no single trigger for a global load balancing operation; each time a new view is distributed, each parent at each level decides locally whether and how to change the position of its line, based on disparity in the load of its two children and whether lines above it have been moved.

In the initial implementation of the algorithm, the algorithm calculated this new line position in the following way. Let the two processors owning areas 1 and 2, as shown in Figure 3.6, be called A and B, respectively. Line LP was the position of the division between the two areas for the last frame, so that area 1 consisted of the interval [L1,LP], and area 2 consisted of the interval [LP,L2]. The work in area 1 for the last frame was $w1$, and the work in area 2 was $w2$. M1 and M2 are the new positions where lines L1 and L2 have already been moved for the next frame by parents at a higher level in the load balancing tree. Let the parent of processors A and B be processor C. Processor C must determine where to place line MP so as to make the balance between areas 1 and 2 as equal as possible for the next frame. Already complicating matters is the fact that the boundaries of areas 1 and 2 have been moved at higher levels in the load balancing tree: they are now [M1,LP] and [LP,M2]. These areas may or may not intersect the old areas 1 and 2. In this case, they do intersect: one is a subset, and one is a superset, of the old area.

Processor C has only the following information in order to determine the appropriate position for line MP: the positions of L1, LP, L2, M1, and M2, and the values $w1$ and $w2$. The values $w1$ and $w2$ are scalars measuring the total work in areas 1 and 2, respectively. These two scalars do not give any detail about the distribution of work density within areas 1 and 2; a processor can calculate what the average density is, but it has no information about particularly dense or sparse areas. The initial implementation of the algorithm made the assumption that the density was a linear function of X position (or Y position when balancing horizontal lines) between L1 and L2.

Assuming that the density function for the region [L1,L2] is:

$$d(X) = mX + b$$

m and b can be found in the following way: Call $d1$ the average density to the left of LP, and $d2$ the average density to the right of LP. The values $d1$ and $d2$ may be found as follows:

$$d1 = \frac{w1}{\mathrm{LP} - \mathrm{L1}}$$

$$d2 = \frac{w2}{\mathrm{L2} - \mathrm{LP}}$$

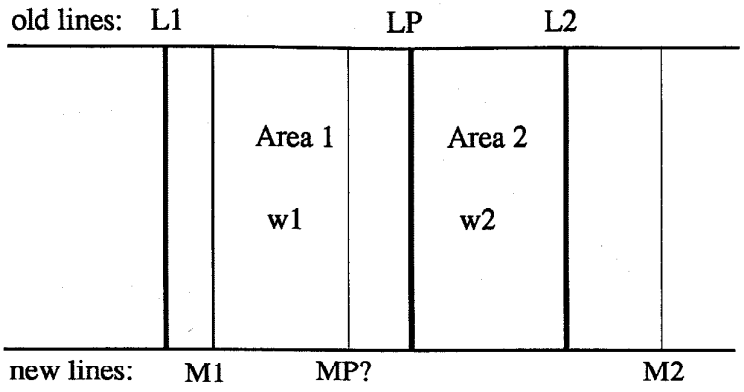


Figure 3.6: Determining Placement of a Load Balancing Line

Now since

$$\int_{\mathrm{L1}}^{\mathrm{LP}} mx + b \quad dx = w1$$

and

$$\int_{\mathrm{LP}}^{\mathrm{L2}} mx + b \quad dx = w2$$

as long as L1 $\neq$ LP $\neq$ L2, which the implementation takes care to assure, m and b can be found as:

$$m = 2\frac{d2 - d1}{\mathrm{L2} - \mathrm{L1}}$$

$$b = d1 - \frac{1}{2}m(\mathrm{LP} + \mathrm{L1})$$

If m is zero, then the density from M1 to M2 is constant, and MP is placed at (M1 + M2)/2. Otherwise, if m is not zero, the following is done: let $v1$ and $v2$ be the amounts of work that are estimated to be in the new areas 1 (i.e., [M1,LP]) and 2 (i.e., [LP,M2]), respectively:

$$v1 = \frac{1}{2}m(\mathrm{MP}^2 - \mathrm{M1}^2) + b(\mathrm{MP} - \mathrm{M1})$$

$$v2 = \frac{1}{2}m(\mathrm{M2}^2 - \mathrm{MP}^2) + b(\mathrm{M2} - \mathrm{MP})$$

Setting $v1$ equal to $v2$ yields a quadratic equation; the solution chosen is the root for MP that lies between M1 and M2.

A more advanced method to load balance based on center of mass will be discussed in Section 6.1.1 of this paper.

3.3.4 Heuristics When an Informed Line Placement Decision Is Impossible

What happens if M1 > L2, or M2 < L1? This may happen if lines M1 or M2 have already been moved by a processor higher in the load balancing tree. Then the old and new areas 1 and 2 do not intersect, and the processor trying to make the line placement decision has no information at all about the local work density inside the new areas. In this case it must resort to a heuristic to guess at a reasonable load balance. Such a heuristic that was empirically found to be useful is to set MP to some position XP such that

$$\frac{\mathrm{XP} - \mathrm{M1}}{\mathrm{M2} - \mathrm{XP}} = \frac{\mathrm{LP} - \mathrm{L1}}{\mathrm{L2} - \mathrm{LP}}$$

That is, set it so that the ratio of the widths of the left and the right sides for the current frame are the same as it was for the last frame. Solving for such a position XP yields:

$$\mathrm{XP} = \frac{\mathrm{M1}(\mathrm{L2} - \mathrm{LP}) + \mathrm{M2}(\mathrm{LP} - \mathrm{L1})}{\mathrm{L2} - \mathrm{L1}}$$

In fact, it was found that using this heuristic when a weaker condition held true helped the load balance on average. That condition was:

$$(\mathrm{M1} > \mathrm{LP}) \wedge (\mathrm{M2} > \mathrm{LP}) \wedge (\mathrm{M1} < (\mathrm{LP} - 2(\mathrm{LP} - \mathrm{L1})) \wedge (\mathrm{M2} > (\mathrm{LP} + 2(\mathrm{L2} - \mathrm{LP}))$$

That is, whenever M1 or M2 has moved greatly (by either (LP−L1) or (L2−LP), respectively) in either direction, then the load information is inaccurate enough that, on average, the heuristic does better.

Another heuristic that was tried, but was empirically found to be useful only in one case, is to set MP to the center point of M1 and M2:

$$\mathrm{MP} = \frac{\mathrm{M1} + \mathrm{M2}}{2}$$

This heuristic was only helpful when $w1$ and $w2$ were both zero, and the current implementation uses it only in this case. Otherwise, it tended to erase progress toward convergence when the equilibrium point was far from the midpoint, and on average did more harm than good.

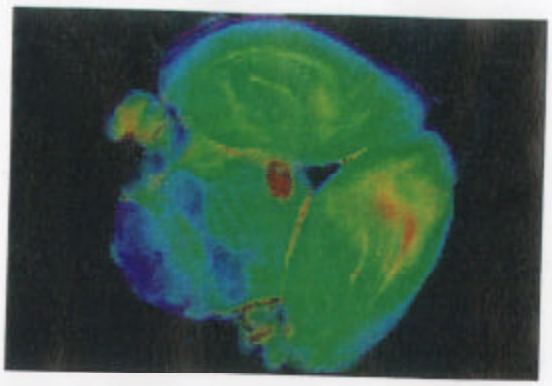
Chapter 4

Example Animated Sequences

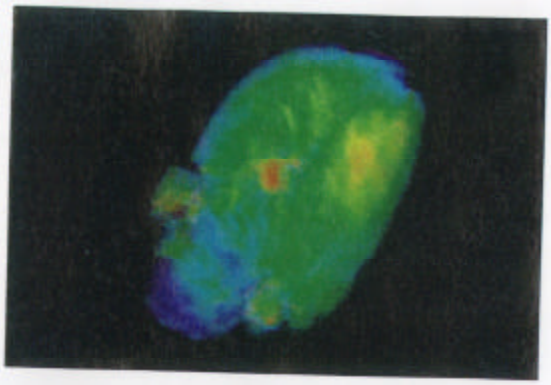
The animated sequences that follow demonstrate the characteristics of the algorithm on several data sets originating from a Magnetic Resonance Imaging (MRI) device. MRI data measures water proton density in biological tissue, resulting in 3D scalar data sets. Although MRI usually generates rectilinear data, the data used in all of these sequences was resampled into unstructured tetrahedral voxels to demonstrate the algorithm's ability to rendering untructured data. All the sequences, except the first one, were created using either 256 or 512 processors of the Intel Delta Machine; the first sequence was created with a software simulation of the same parallel code running on a Sun workstation. A videotape containing more detail of these sequences is also available.

4.1 Output of Object-Order Composition

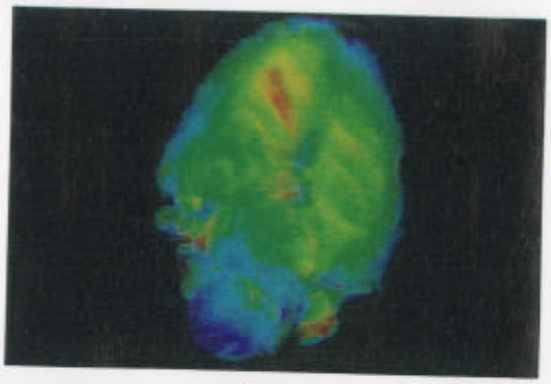
Figure 4.1 shows six frames from an animation rendered from MRI data showing a coronal slab of the brain of an Owl Monkey. This sequence provides an example of object-order composition as performed by our implementation. Notice how the data appears translucent as it is rotated, in keeping with the modeling of a scalar field as translucent material of varying opacity and color, as described in Section 2.1.1.



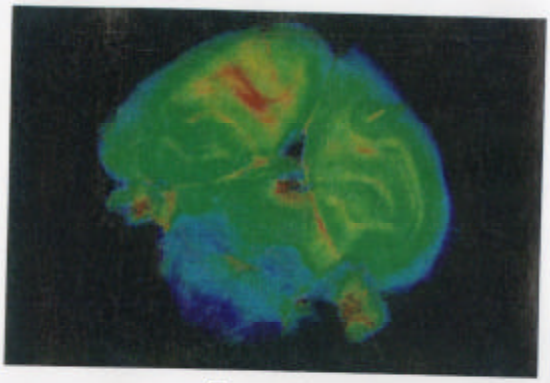
frame 0



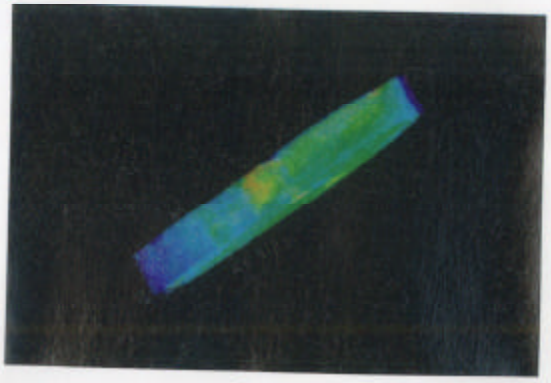
frame 4



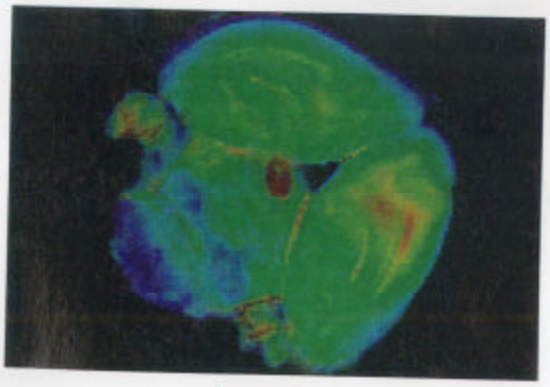
frame 9



frame 13



frame 18



frame 23

Figure 4.1: Coronal Slice of Owl Monkey Brain

4.2 Load Balancing with 256 Processors

Figure 4.2 shows the entire volume of the same brain, rather than just a slab, at low-resolution. The brain is roughly spherical. All voxels in the middle of the color range — the green, blue, and yellow ones — have been made transparent by giving them an opacity of zero. The voxels that remain are the purple outer surface of the brain and some red features in the interior.

Overlaid on the image are gray rectangles representing the areas owned by each of 256 processors. The six frames (numbered 0,3,6,9,12, and 15) display a time sequence of dynamic load balancing. The density of colored area is roughly proportional to the density of workload. Frame 0 has the initial default load balance: note that many processors contain no voxels, and therefore have no rendering work to do. The distribution of processors to areas of high work density improves continually until all processors have nearly equal workload by the last frame, frame 15. The load balancing lines find their equilibrium position in order of their level on the load balancing tree (as was shown in Figure 3.2). Note that the most visible difference between frames 12 and 15 is the position of the lines in each of the extreme corners of the screen — these are some of the lines at the lowest level of the tree.

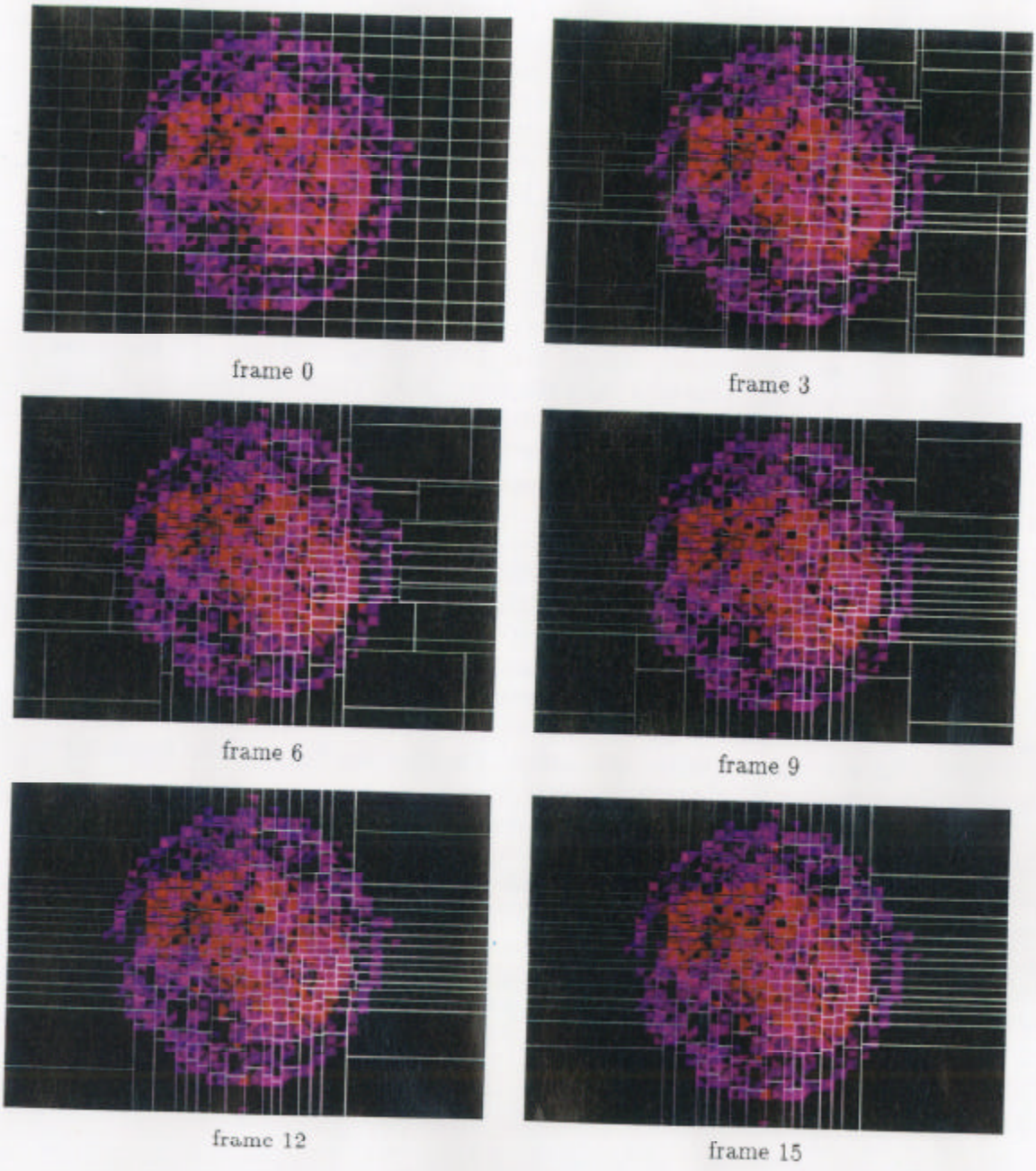


Figure 4.2: Load Balancing with 256 Processors

4.3 Adapting to Gaps with 256 Processors

Figure 4.3 is a sequence rendered from a data set containing just one slice through the data shown in the previous figure (Figure 4.2). Because this is the rendering of a slice rather than a volume, and since the voxels in the middle of the color range (the green, blue, and yellow ones) were removed, gaps appear between the remaining voxels (the red and purple ones). Since Figure 4.2 was a composition of many slices, the gaps were mostly filled in.

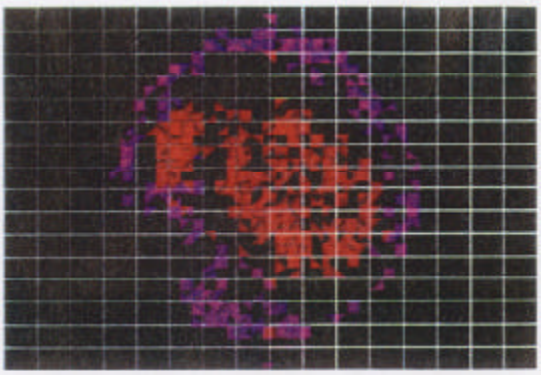
The figure demonstrates that the dynamic load balancing algorithm will achieve good load balance even with data in unusual distributions. The algorithm implicitly works around gaps in the data — not by a search for such gaps, but simply by recursively balancing the hierarchy of lines as described in Section 3.3.3. Note that by the final frame, each rectangle contains a similar amount of colored area, even working around the gaps.

4.4 Following Moving Data

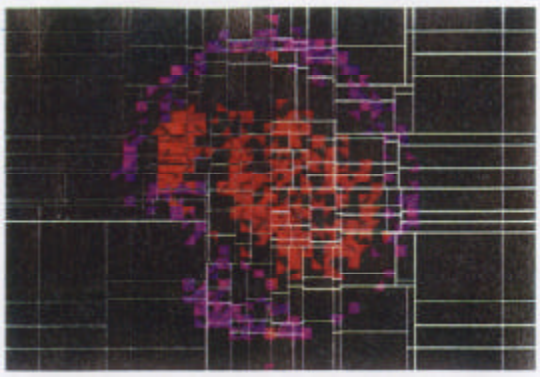
Figures 4.4 and 4.5 show a sequence rendered from data from the MRI scan of a live mouse, rendered with 512 processors of the Intel Delta Machine. These frames were taken from a longer animated sequence of 210 frames. The sequence demonstrates the ability of the load balancing algorithm to track moving data. Between frames 131 and 151, the mouse slowly rolls toward the observer. Between frames 155 and 175, the nose of the mouse turns from the (observer's) left to the right.

Note at all times that the load balancing lines are able to track the movement of the data and maintain balance. The point of worst balance is between frames 167 and 171, where the nose of the mouse has swung quickly around — note that some of the vertical lines on the bottom of the screen are compressed together, instead of being equally spaced. Quick movement of data implies that each processor will have less accurate local load information when it tries to calculate an appropriate balance; in this case, some processors had no local information and had to resort to a guessing heuristic. One way to reduce this effect would be to increase the area about which processors have load information. The algorithm has recovered by frame 175, where it can be seen that the lines are again well spaced.

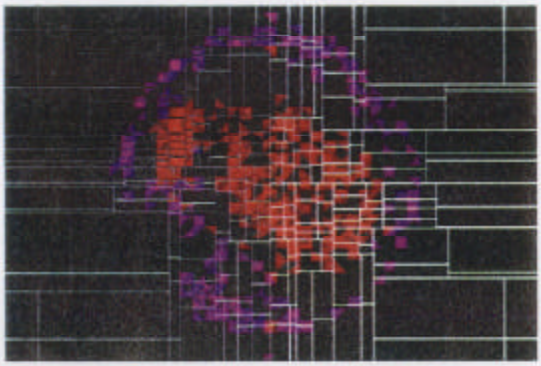
This data set contained approximately one million tetrahedral voxels. For this data set, we achieved average rendering frame rates of 25 seconds per frame. Most of the expense was from communication, and we expect that the future optimization described in Section 7.1.2 would decrease this cost by about 60%, for 512 processors. We expect the new partitioning methods proposed as future work in Section 7.1.1 to further decrease this cost. Furthermore, new types of concurrent architectures are appearing which offer greatly reduced communications costs. We intend to port the current implementation to MIT's J-Machine [Dally89], for which communications costs are at least one order of magnitude less than those for the Intel Delta Machine.



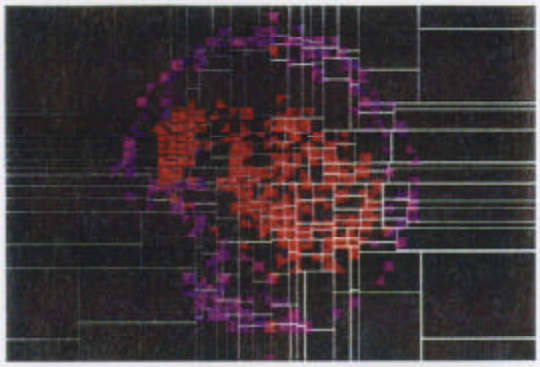
frame 0



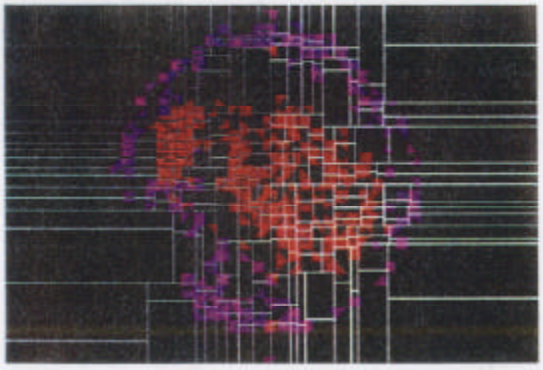
frame 3



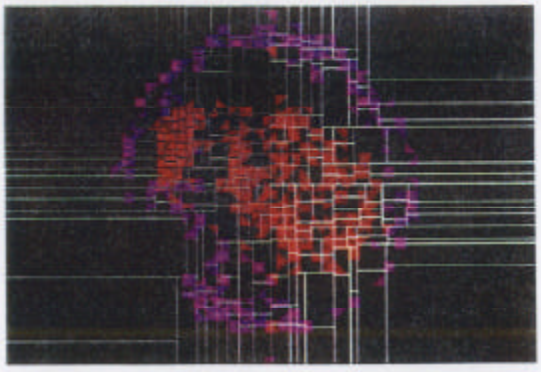
frame 6



frame 9

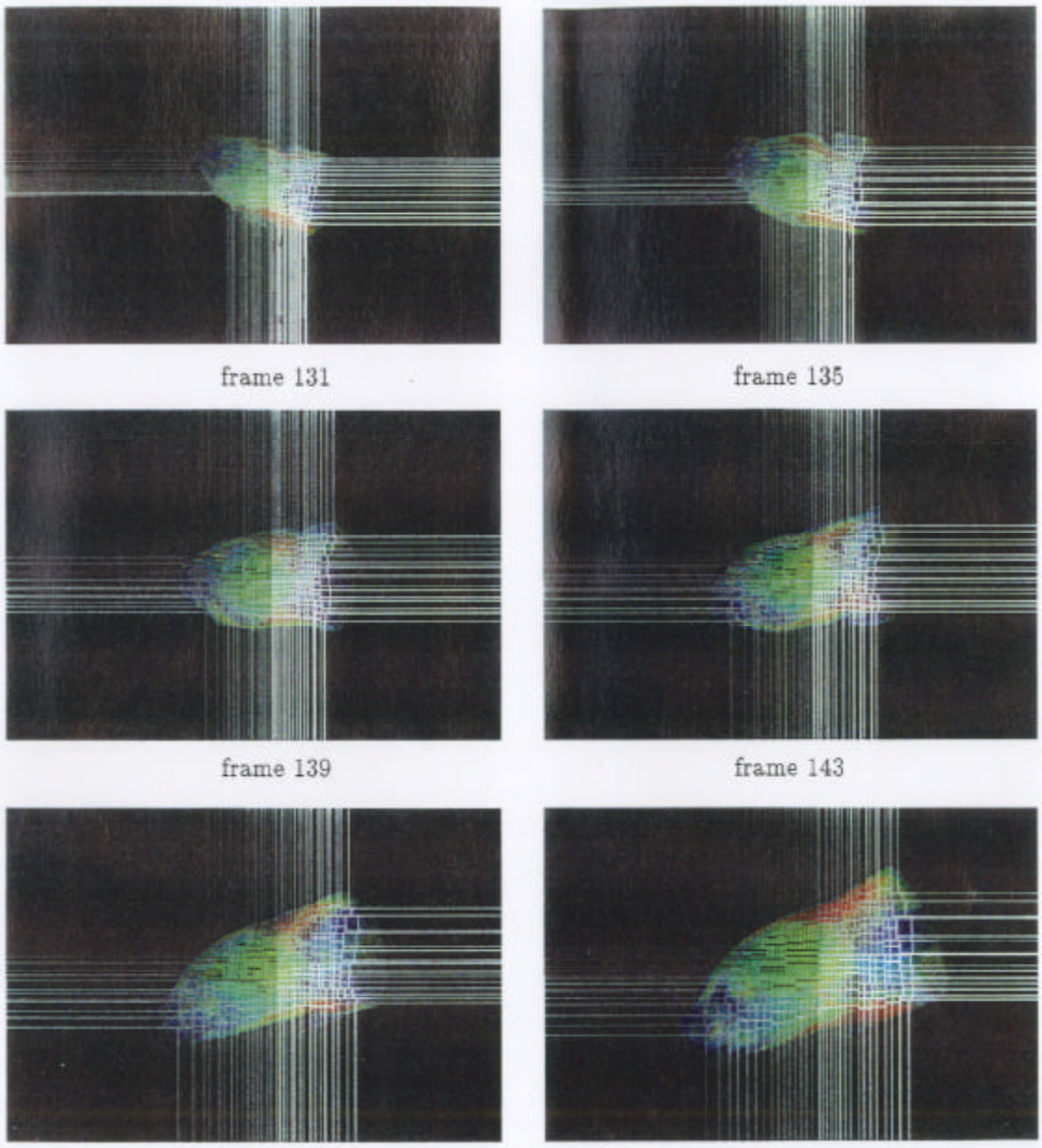


frame 12



frame 15

Figure 4.3: Adapting to Gaps in the Data

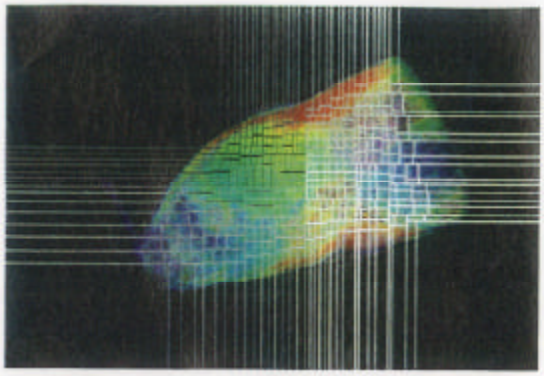


frame 131 frame 135

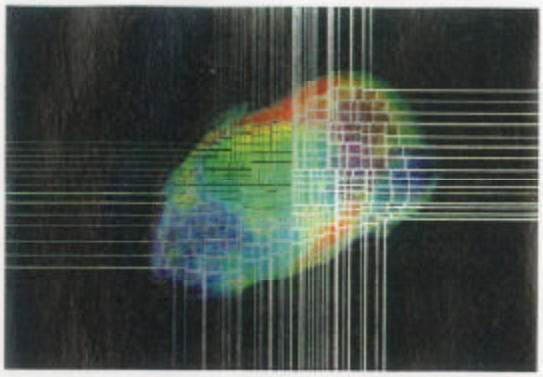
frame 139 frame 143

frame 147 frame 151

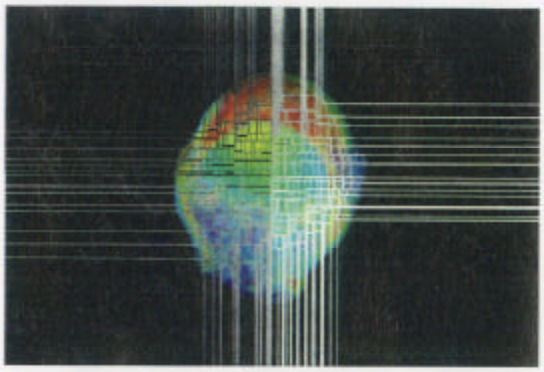
Figure 4.4: Following Moving Data with 512 Processors, Mouse Head



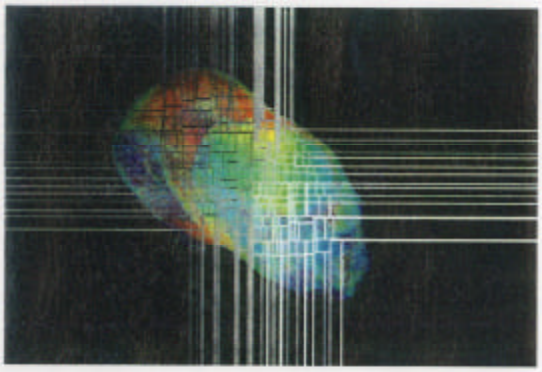
frame 155



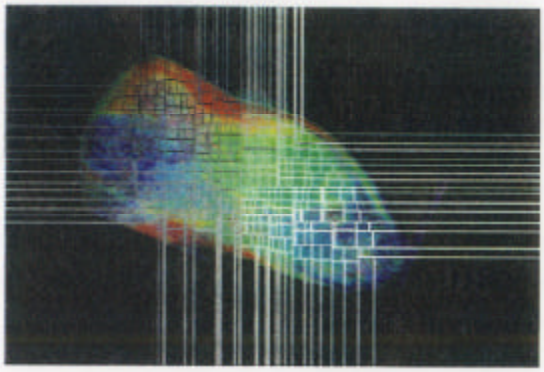
frame 159



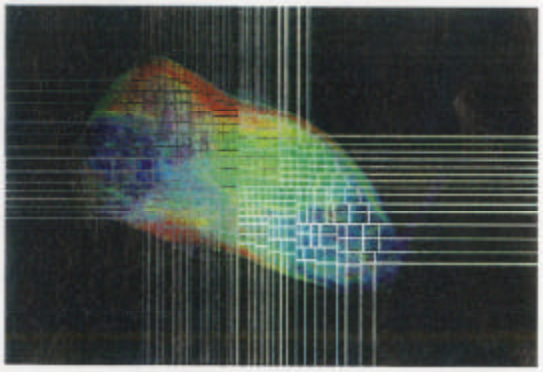
frame 163



frame 167



frame 171



frame 175

Figure 4.5: Following Moving Data with 512 Processors, Mouse Head (Continued)

Chapter 5

Algorithm Performance

5.1 Analysis

5.1.1 Load Balance and Computational Scalablity

Assuming good load balance can be maintained, the algorithm is scalable as far as computation and memory use are concerned. The total amount of work and the total number of voxels will remain nearly constant as the number of processors increases. Therefore, for N processors, the amount of work that each processor has will decrease as $\frac{1}{N}$. There are slight increases in the number of voxels as number of processors increases, because the increase in lines dividing the screen requires the generation of shadow copies of voxels. However, this number is quite small compared to the total number of voxels as long as the size of a voxel remains small relative to the portion of screen area owned by each processor.

One condition under which which good load balance could not be maintained would be if the number of voxels were much less than the number of processors — then there would be less than one voxel per processor, and some processors would necessarily be idle; however, this will not be the case for the foreseeable applications of the algorithm.

A more common condition under which load balance could not be maintained would be during rapid movement or rotation of the viewpoint of the observer. Because the overlap between the volume owned by a processor in the current image space and what it will own in the next image space may be small or nonexistent, the processor may have little local load information on which to base load balancing decisions. It is assumed by the principle of *viewpoint coherence* that viewpoint changes will be limited. The effect may increase with increasing numbers of processors, because the dimensions of areas owned by processors will decrease, while typical *per-second* rates in viewpoint will remain constant. Frame rate increases due to the use of more processors may offset this by decreasing *per-frame* rates of viewpoint change, but frame rate increases greater than perhaps 60Hz would be of limited use. Some suggestions to increase the area for which processors have accurate load information, which could offset this effect, are mentioned in the future work chapter, Chapter 7.

5.1.2 Communications Scalablity

Communications performance in the current algorithm is not expected to scale linearly. Nearly all of the communications generated by the algorithm consists of hopping voxels down the voxel

distribution tree. The number of hops generated can be analyzed in the following way:

Figure 5.1 depicts a voxel slowly traversing the screen; during the traversal, the voxel must cross two lines of level 2, and one line of level 0. (This voxel could alternatively be considered as the average voxel, which at any time would be making the average of these line crossings.) To cross the line at level 0, which is at the top of the distribution tree, incurs more hops on average than to cross a line at level 2. More precisely, to cross a line at even-numbered level L, when there are N processors, requires $\frac{1}{2}(\log N - L)$ hops (for $\log N$ odd). The factor of $\frac{1}{2}$ arises because, on average, half the time that a voxel must be sent down to the next level, the current owner will also be a valid owner at the next level, eliminating the need for a physical communication; instead the voxel is moved onto another queue on the same processor. (For a processor to be a valid owner at a given level means that it knows the position of the next line that the voxel must be compared against at that level; recall that processor knowledge of line positions was shown in 3.4).

As the voxel crosses the screen, it will have to cross $2^{\frac{L}{2}}$ lines at each even-numbered level L, yielding a total number of hops generated of:

$$\sum_{L=0...2...(\log N)-1} 2^{\frac{L}{2}} \frac{1}{2}(\log N - L) \qquad \text{for } \log N \text{ odd.}$$

This is a superlinear increase in N, so communications load will not be expected to scale linearly with use of more processors, as computational load does. Chapter 6 of this paper describes some modifications made to the basic algorithm to improve this; Chapter 7 suggests further future improvements.

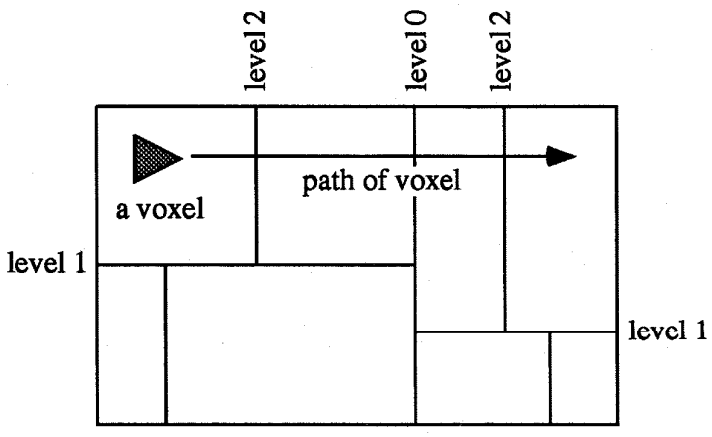


Figure 5.1: A Voxel Slowly Traversing the Screen Crosses Several Lines

5.2 Empirical Study - Delta Machine Experiments

5.2.1 Load Balancing of Computation and Memory

The voxels making up the data account for the vast majority of memory use by the algorithm; the amount of computational work done by each processor is also roughly proportional to the number of voxels they hold. Looking at the number of voxels held by each processor is a way to measure the load balance of both memory and computation simultaneously.

Perfect load balance is defined as the condition that the maximum number of voxels held by any processor is equal to the average held by each processor this implies that all processors have exactly the same number of voxels. A good load balance therefore requires that the maximum and average number of voxels be as close to each other as possible.

Figures 5.2, 5.3, 5.4, and 5.5 show statistics taken during the rotation of a data set consisting of an MRI scan of a monkey brain, such as that shown in Figure 4.2, but at four times the resolution. The non-empty cells in the data make up a roughly spherical volume which is rotated around a point slightly removed from its center of mass. For runs of 32, 64, 128, and 256 processors, the figures display the maximum number of voxels held by any processor, and the average number held by each processor, at each timestep. The figures differ in the rate at which the data set is rotated — in the four figures the data is rotated 1, 5, 10, and 30 degrees per frame, respectively. The reader may compare the distance between "max-32-1" to "avg-32-1" (in Figure 5.2) to see how close to optimal load balance a run of 32 processors could maintain at a rotation of 1 degree per frame (and likewise "max-64-1" to "avg-64-1", etc.). Note also, from frames 0 to 5, the sharp decrease of the maximum lines for all numbers of processors as the algorithm converges from the initial regular distribution of processors to an balanced distribution, as was illustrated in Figure 4.2. The average lines go up slightly as the total number of voxels increases when the converging lines move into the data and shadow copies must be generated.

For low rates of rotation (e.g. 1 degree per frame, Figure 5.2), the algorithm maintains excellent load balance for all numbers of processors — the maximum number of voxels held by any processor is nearly constant and is only about 125% of the average number, for all number of processors. Since the number of voxels is roughly proportional to computation time, this means that the busiest processor would only take 125% as long as the average processor to finish its computation. By this measure, the 256 processor runs perform as well as the 32 processor runs, showing that the algorithm can maintain near-ideal load balance as long as data movement is suitably constrained. Note that the maximum value for 32 processors is about twice the value for 64 processors, which is twice that for 128 processors, which is in turn about twice that for 256 processors.

For higher rates of rotation (e.g. 5 and 10 degrees per frame, Figures 5.3 and 5.4), the relationship between image space and data space is changing more quickly. This makes the overlap between the volume owned by a processor in the current image space, and the volume owned by that processor in the next image space, smaller. Processors therefore have less accurate information about what the distribution of work will be within their volume for the next frame. This lack of information forces them to resort to a guessing heuristic more often, decreasing load balance performance. Note that the maximum lines are no longer constant but sometimes waver as the algorithm tries to maintain balance. However, at 5 degrees per frame, for 32 and 64 processors, the maximum number for any processor was, on average, still only 170% of the

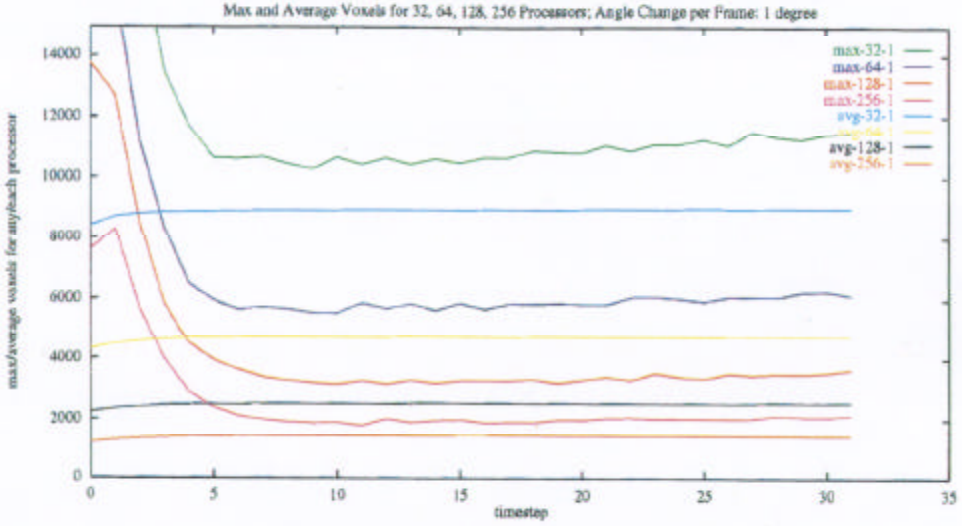


Figure 5.2: Maximum and Average Numbers of Voxels per Processor for 32, 64, 128, and 256 Processors, 1 Degree Rotation per Frame

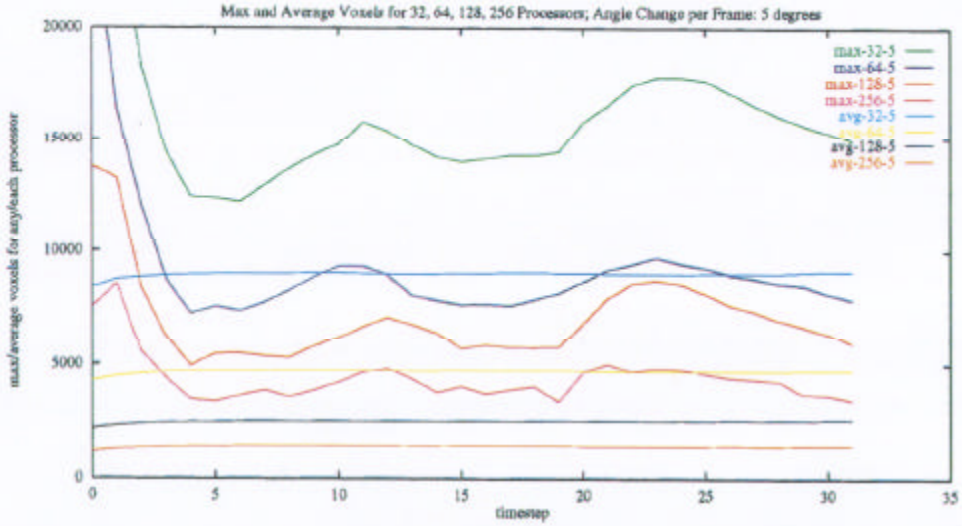


Figure 5.3: Maximum and Average Numbers of Voxels per Processor for 32, 64, 128, and 256 Processors, 5 Degrees Rotation per Frame

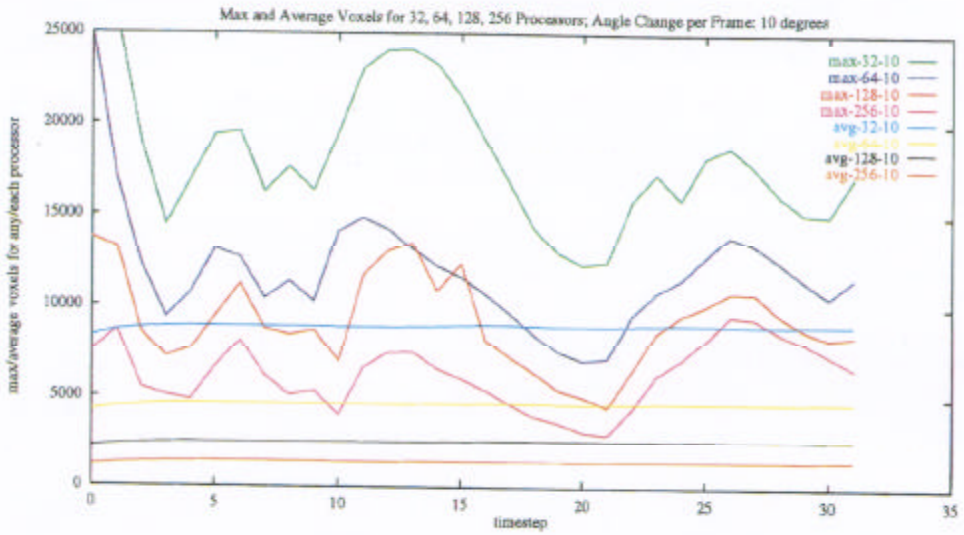


Figure 5.4: Maximum and Average Numbers of Voxels per Processor for 32, 64, 128, and 256 Processors, 10 Degrees Rotation per Frame

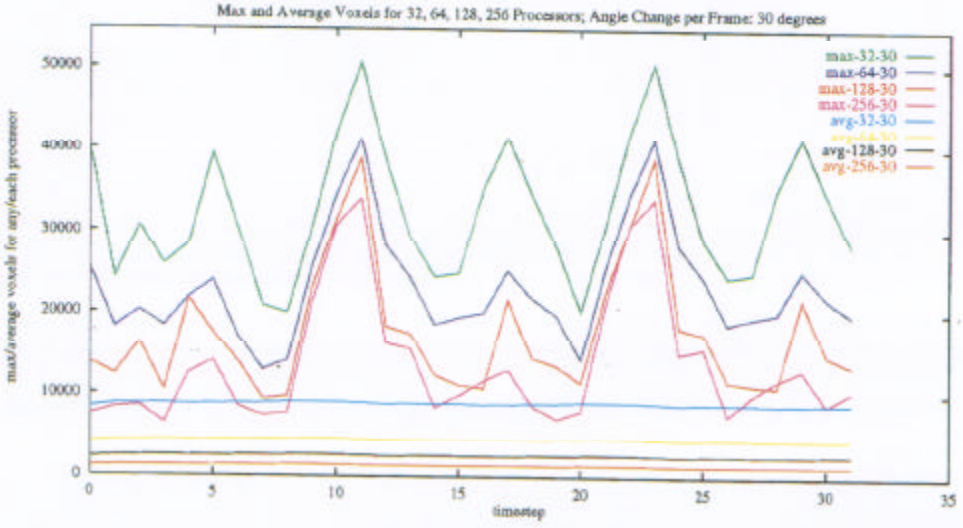


Figure 5.5: Maximum and Average Numbers of Voxels per Processor for 32, 64, 128, and 256 Processors, 30 Degrees Rotation per Frame

average number; and, for 128 and 256 processors, about 300%. At 10 degrees per frame, the maximum was about 190% for 32 and 64 processors; and about 400% for 128 and 256 processors.

In the worst case, at 30 degrees per frame (Figure 5.3), the difference between average and maximum numbers of voxels varies between 125% and 2600%, depending on the particular orientation of the data set and the number of processors. The worst point occurs at timestep 11. (A similar point occurs again after 360 degrees of rotation, i.e. 12 timesteps of 30 degrees each, at timestep 23.) At this point, the maximum number of voxels for 256 processors is about 2600% of the average number. At this worst point, many processors have little information about the local work density and are using the heuristic to guess at an appropriate balance; this, as expected, does not work as well as when there is more information available.

The overall conclusion to be drawn from these four figures is that below a certain threshold level of rotation, between 1 and 5 degrees per frame, the algorithm can maintain excellent load balance; at higher rates of rotation this gracefully degrades, with larger numbers of processors (with their relatively smaller areas for which they have load density information) first affected. At about 30 degrees per frame, most processors are using the guessing heuristic rather than a sound calculation of appropriate load balance; predictably, load balancing performance degrades rapidly.

5.2.2 Communications Performance

Since the hopping of voxels down the distribution tree accounts for the vast majority of communication, a simple way to measure communication and its scalability is to count the number of hops each processor performs.

Figures 5.6, 5.7, 5.8, and 5.9 also show statistics taken on the same rotating brain volume as the statistics in the previous subsection. The figures display the maximum number of hops performed by any processor, and the average number performed by each processor, for 32, 64, 128, and 256 processors. Again, the four figures differ in the rate at which the data set is rotated — they show 1, 5, 10, and 30 degrees per frame, respectively.

In order for the algorithm to scale linearly with number of processors, the number of hops (communications) per processor should halve with each doubling of processor number. This is not the case, however, as predicted by the analysis in Section 5.1.2, since, as the number of processors increases, the total number of hops increases more rapidly. Therefore, communication is the expected bottleneck as the number of processors is increased. This is confirmed in Figures 5.6 through 5.9, where the reader will note that the average number of hops per processor does not halve with each doubling of processor number.

As for communications load balance, the maximum number of hops stays near the average number at low rates of rotation; for instance, at rates of 1 and 5 degrees per frame (Figures 5.6 and 5.7), the maximum is never more than 200-300% of the average, and is usually with 120%. At high rates of rotation, such as at 10 and 30 degrees per frame (Figures 5.8 and 5.9), the difference between maximum and average increases for certain orientations of the data (orientation of the data may be correlated to timestep since the data is rotated at a constant rate). These periods of communication imbalance were generally associated with periods of computational imbalance. This orientation dependence is again because the voxels in the data set make up an irregular shape which only approximates a sphere, and the data is rotated around a point slightly removed from its center of mass in this experiment.

Overall, although no special consideration was taken to balance communication — only the

number of voxels is explicitly balanced — a well-balanced communication load is sustained for similar rates of data rotation as those for which computational load balance is sustained. Communication load, however, is not scalable, as computational load is: as the number of processors increases, the amount of communications required increases more quickly, as predicted by the analysis in Section 5.1.2. The Optimizations chapter of this paper, Chapter 6, describes some changes made to the basic algorithm to improve this. The Future Work chapter, particularly Sections 7.1.2 and 7.1.3, makes further suggested improvements.

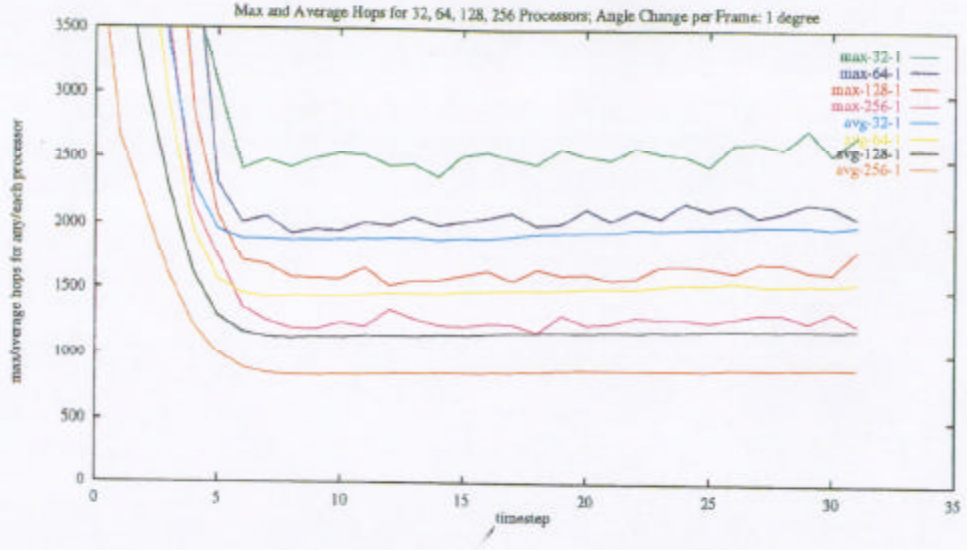


Figure 5.6: Maximum and Average Number of Hops per Processor for 32, 64, 128, and 256 Processors, 1 Degree Rotation per Frame

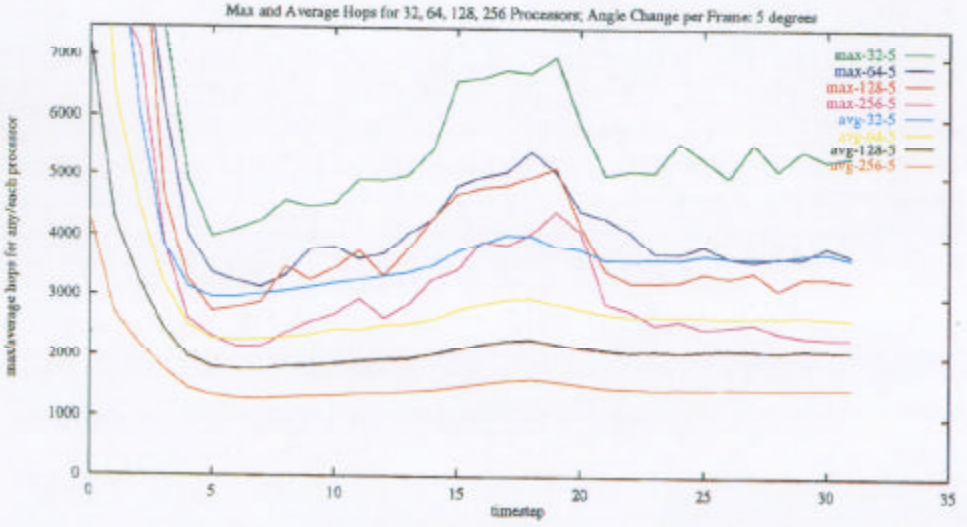


Figure 5.7: Maximum and Average Number of Hops per Processor for 32, 64, 128, and 256 Processors, 5 Degrees Rotation per Frame

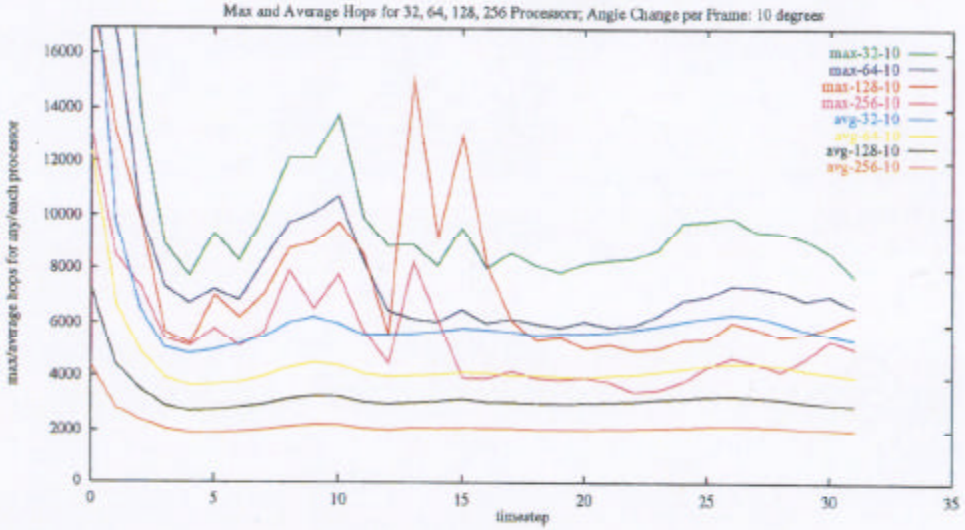


Figure 5.8: Maximum and Average Number of Hops per Processor for 32, 64, 128, and 256 Processors, 10 Degrees Rotation per Frame

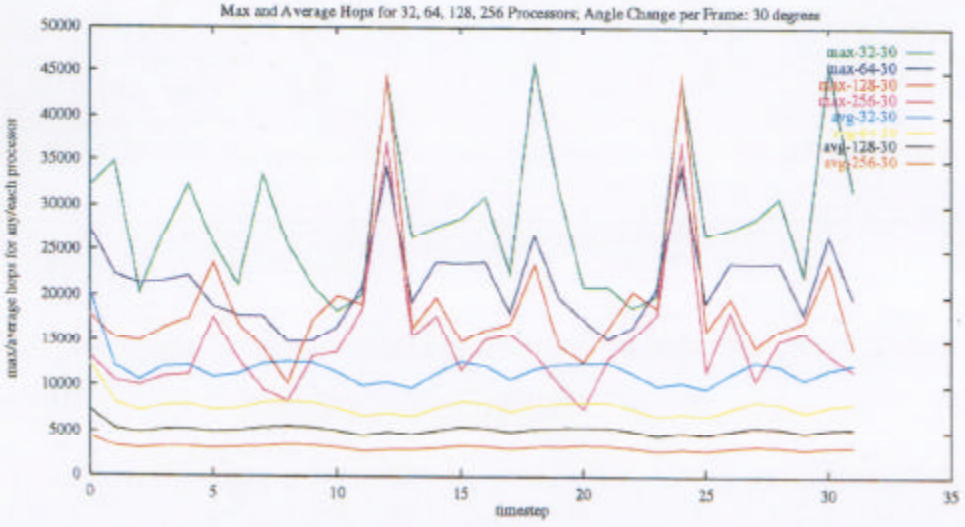


Figure 5.9: Maximum and Average Number of Hops per Processor for 32, 64, 128, and 256 Processors, 30 Degrees Rotation per Frame

5.2.3 Combined Timing Results

Wall clock timings of the same experiment as in the previous two subsections are shown in Figure 5.10, where it can be seen that time per frame is not halved with a doubling of processor number. The figure shows the average wall clock time per frame for various numbers of processors and various rates of rotation of the data. A perfectly scalable algorithm would produce straight lines with a slope of negative one, which the original algorithm does not do; this is directly attributable to the predicted and empirically observed communications bottleneck.

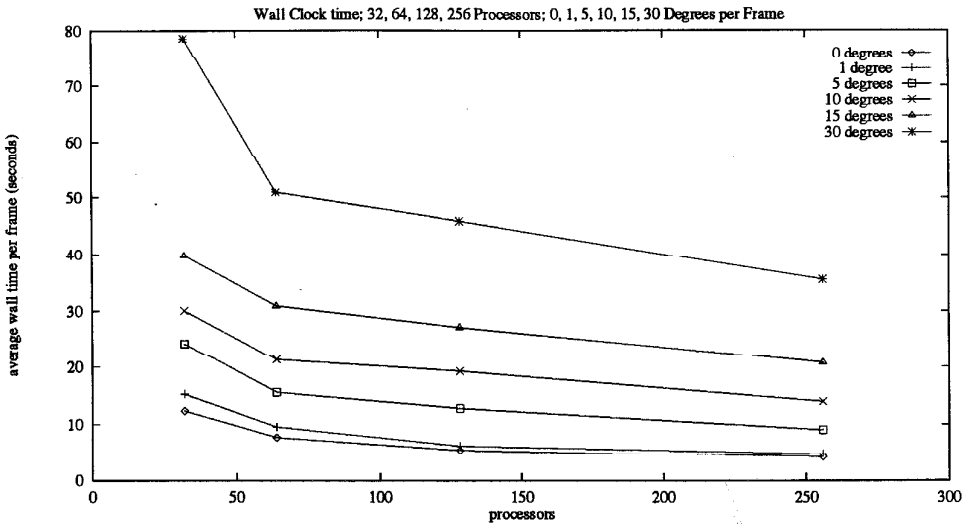


Figure 5.10: Wall Clock Timings on Intel Delta

Chapter 6

Optimizations

This chapter explains a variety of optimizations that were applied to the algorithm described above and included in the current implementation. We first describe optimizations that are original contributions of this thesis, and then describe some optimizations in common practice which were also used in the current implementation. Most of the novel optimizations are qualitative improvements to the load balancing or distribution algorithms; the standard optimizations are all methods to improve performance of specific tasks in the overall algorithm.

6.1 Novel Optimizations

6.1.1 Center of Mass versus Simple Count

Perhaps the most important addition to the original load balancing algorithm was the use of a *center of mass measure* for work rather than the simple voxel count described in Section 3.3.3. This allowed a qualitatively superior load balance: not only did it allow more accurate determination of ideal load balance for motionless data, it allowed the *anticipation to the next frame* of the changing loads incurred by data movement, so that moving data could also be precisely balanced; this removed a time lag from which the voxel count method had suffered.

The reader will recall that the center of mass of a collection of masses m_i at positions x_i is the vector:

$$\vec{\mathrm{CM}} = \frac{\sum_i \vec{x_i} m_i}{\sum_i m_i}$$

The current implementation used the center of each voxel as the position x_i and gave each voxel a mass of one (1). Another possibility would be to use the projected surface area of the voxel, since this is proportional to the number of pixels that it effects. Knowing the numerator and denominator in the above equation for two regions, one can easily find the center of mass of their union. In this way, the center of mass information is passed up the load balancing tree.

Using the center of mass of the voxel distribution, instead of the common alternative of simply *counting the voxels* in each area, allows a qualitatively superior load balance: first, because the center of mass more accurately describes the density distribution than a simple count, it allows more accurate determination of ideal load balance for motionless data; furthermore, because the center of mass is a vector, it can be matrix-multiplied by the difference between the last and current view matrices. This effectively *anticipates to the next frame* the change in load density

incurred by data movement, so that moving data can also be precisely balanced, without a chronic time-lag.

To load balance with the center of mass information, the algorithm multiplies the center of mass by the difference between the current and next view matrices, adjusts for the changes in M1 and M2, and places its guess at the position of the extrapolated center of mass. Since the center of mass, by definition, is at the center of the density distribution, this should put half the work on each side of each line.

6.1.2 Guessing Correct Final Destination of Voxels

A major source of communication load is the multiple hops required by voxels to travel down the tree of lines subdividing the screen. The algorithm already avoids many hops by checking whether the current processor happens to know the position of the next line down the load balancing tree. If this is the case, the voxel is not sent off to another processor that knows the position, but is tested against the line in the current processor.

Many more hops can be avoided by sending each voxel to the proper owner as high up the load balancing tree as possible. If the proper owner of a voxel receives it, then the voxel can travel all the way down the load balancing tree with no further communications. Although there is no global information on the mapping between regions of space and processors, processors can make a guess. The farther down the load balancing tree a voxel is, the closer the guess can be. The statistics in Figure 6.1 illustrate the effect of this optimization for 256 processors and 5 degrees of rotation per frame.

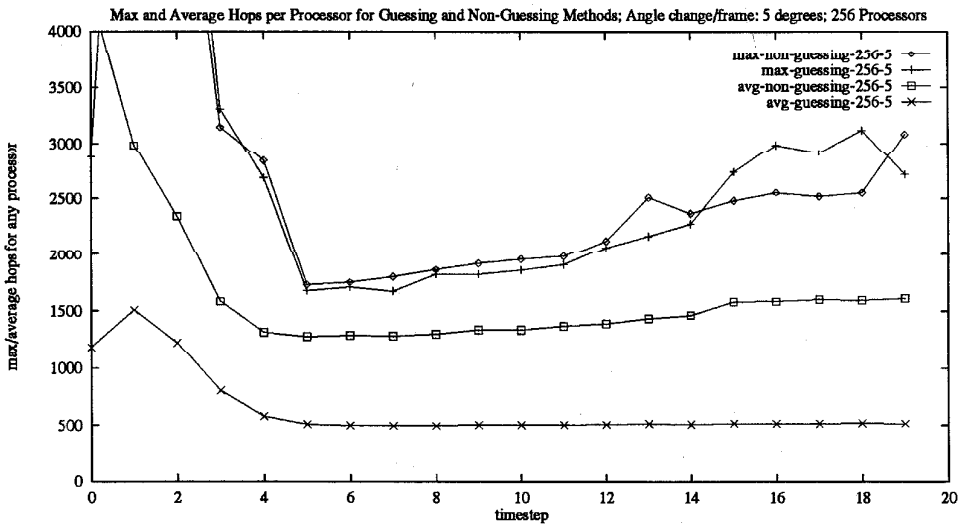


Figure 6.1: Guessing versus Non-Guessing Algorithms, 256 processors

The experiment shown in the figure compares the original *non-guessing* algorithm with the

modified *guessing* algorithm. Whereas the maximum for the guessing algorithm is only slightly below that of the non-guessing algorithm, the average for the guessing algorithm is less than 40% of the average for the non-guessing algorithm. This means that the average processor is generating less than 40% as much communication. Since communication was the bottleneck in the algorithm, this represents a significant performance improvement.

6.1.3 Threshold for Line Movement and Damping of Line Movement

In the first movies made by the Delta Machine implementation of the algorithm, it was apparent that, even when the data was stationary, the lines of load balance would oscillate indefinitely around their equilibrium point, without improving the balance. Every change in the position of the lines incurs communication expense. To eliminate this problem, the following two modifications were introduced to the load balancing algorithm:

If the difference between work on each side of a given line was below a threshold — a percentage of the average between them, called t), then the position of the line would not be changed.

A fixed or dynamic percentage may be chosen by empirical study, or by comparative analysis of communication costs and the benefits of precise balance. In this work an empirically optimized, fixed percentage was used: a line would not move if the work on each side was within 15% of the average. This percentage will be implementation and data dependent, however.

Further control of oscillation is achieved by damping the change in the load balancing lines.

Recall that XP was the position for the load balancing line which kept the widths of the left and the right areas in Figure 3.6 in the same ratio that they were for the last frame:

$$\frac{\mathrm{XP} - \mathrm{M1}}{\mathrm{M2} - \mathrm{XP}} = \frac{\mathrm{LP} - \mathrm{L1}}{\mathrm{L2} - \mathrm{LP}}$$

Assume that NP is the position the algorithm has found as its best guess for optimum balance. A damping coefficient d between 0 and 1 is chosen, and MP, the new position of the load balancing line, is set to:

$$MP = XP + d(NP - XP)$$

This was empirically found to eliminate oscillation; we found that a coefficient d of 0.65 to be most effective on average in a wide range of situations. However, dynamic methods to set the coefficients t and d could be invented and might be an interesting topic further study.

6.2 Standard Optimizations

There are several optimizations to volume rendering which are standard practice. We have applied the following optimizations with positive results.

6.2.1 Keeping Data Only in Current Screen Coordinates

This optimization aims to reduce the length of communicated messages by reducing the storage requirements of an individual voxel. In the original implementation, voxels included the coordinates of a voxel's center and vertices in both object space and image space coordinates — the "long voxel" method shown in Figure 6.2. In this method, the original center and vertices of the

voxel are stored, and with each change of the transformation between object space and image space, the center and vertices of the voxel in image space are calculated from the original values. This approach was taken for simplicity and to avoid the buildup of errors that could result from storing the center and vertices only in the current image coordinates — the "short voxel" method shown in Figure 6.2. In this method, processors keep only one copy of the center and vertices of the voxel and transform them at each frame by the difference between the current and last transformation matrices. This method introduces some complication applying perspective, as the perspective must be removed, the transformation applied, and the perspective reapplied.

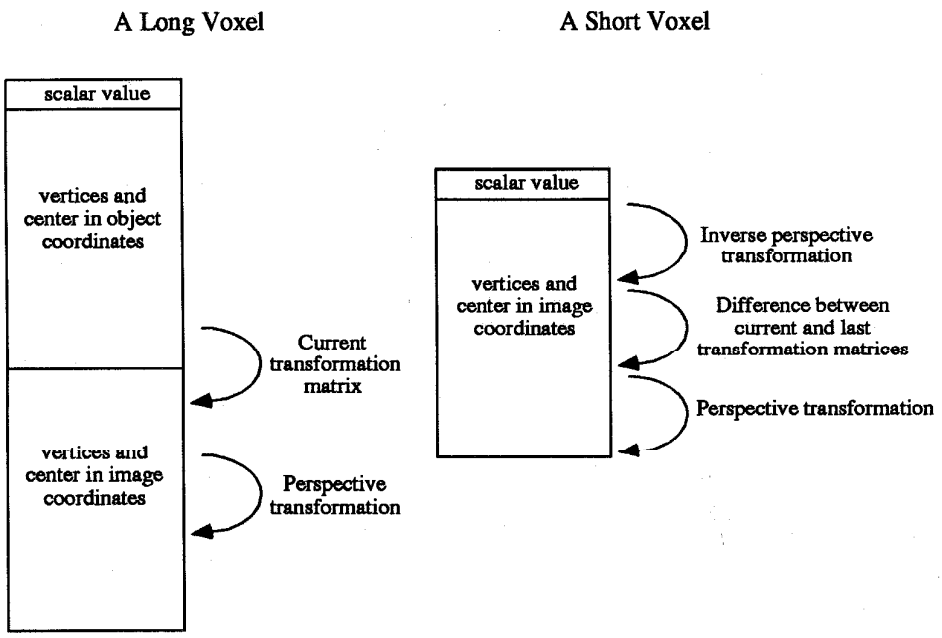


Figure 6.2: Long and Short Voxels

The short voxel method does slowly accumulate errors as the voxels are subjected to repeated transformation, but we found empirically that for the number of views that were generally rendered of a single data set in the course of this work (several hundred) these have not been significant; perhaps this would be different for greater number of repeated transformations.

Using the "short voxel " method resulted in improved wall clock times per frame, because of the decreased communications costs. Figures 6.3 and 6.4 show average wall clock times for runs where a data set was turned by 0 degrees and 15 degrees per frame, respectively. The two figures show, for 64, 128, and 256 processors, the average wall clock time per frame using long voxels and short voxels. When the data set is turned 0 degrees per frame, the only voxels that need to be communicated are the shadow copies; when the data is turned 15 degrees per frame, many more voxels need to be communicated. Predictably, therefore, when more voxels are communicated, the short voxel method shows a greater performance improvement; the reader

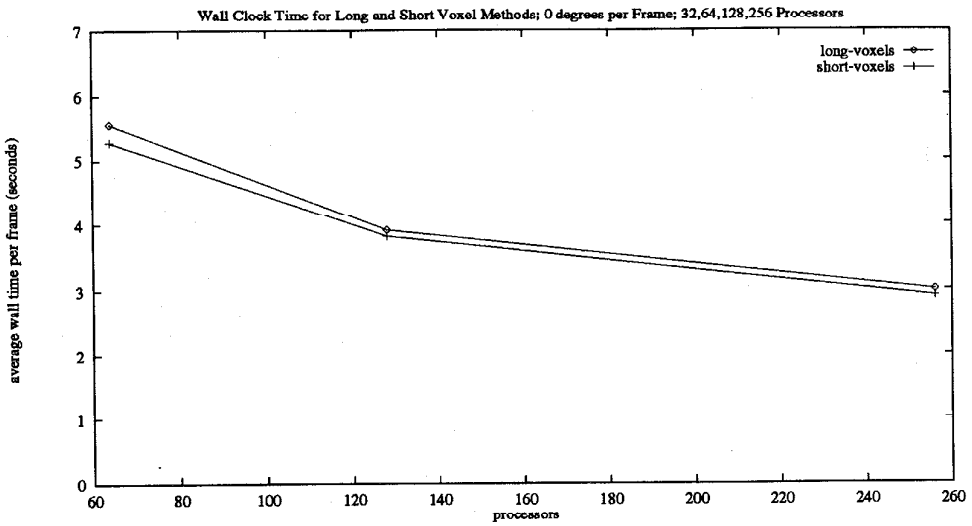


Figure 6.3: Long and Short Voxels at Rotation of 0 Degrees per Frame

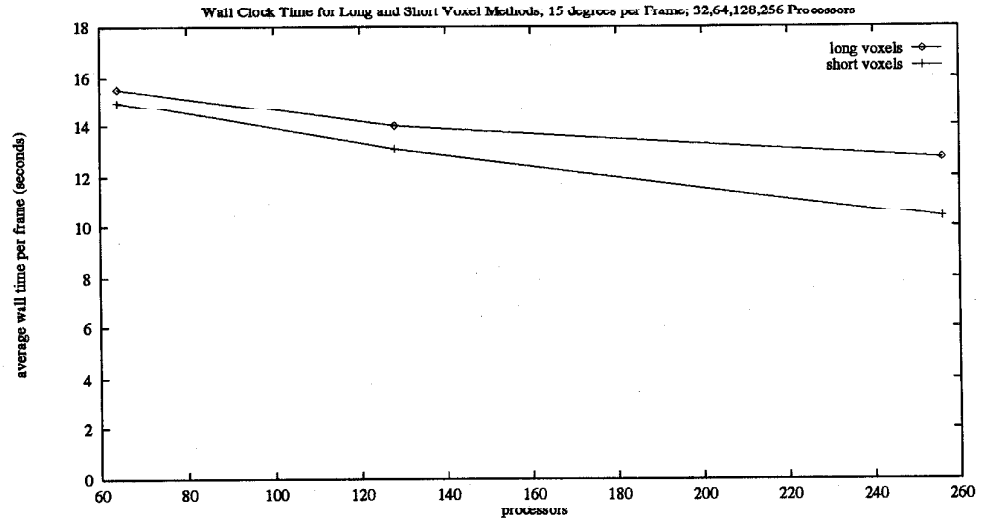


Figure 6.4: Long and Short Voxels at Rotation of 15 Degrees per Frame

will note that the improvement using short voxels over long voxels is more in Figure 6.4 than in Figure 6.3. Furthermore, the improvement is more for greater numbers of processors than for smaller numbers — again because the communications costs for greater numbers of processors becomes more a greater part of the wall clock time relative to the computation. With 256 processors, at 15 degrees per frame, the improvement is significant — about a 20% reduction in wall clock time on the Delta Machine.

6.2.2 Free Voxel List

A common trick to remove the overhead of using "malloc" and related C functions when frequently allocating and deallocating a type of object is to do a one-time allocation of a large list of these objects, take them from the list when needed instead of allocating new ones, and put them back on the list instead of deallocating them. We maintained such a "free voxel" list and found that it offered a slight performance improvement in all cases tried.

6.2.3 Integer Arithmetic

The use of integer arithmetic wherever possible instead of floating point arithmetic is another way to improve performance. In the algorithm, there were two places where this would have a significant effect and was possible: first, in the calculations of opacity and color during composition; and second, in the matrix and vector calculations. Only the first was included in the Delta Machine implementation and did afford a small performance improvement. The second was not implemented because the performance bottleneck in the Delta implementation turned out to be in communication rather than computation. Software simulation of a multiple processor machine on a Sun workstation showed more pronounced performance improvement because the software simulation did not require the communication delays of the real Delta Machine. In a future J-Machine implementation it would probably be useful to avoid floating point arithmetic in both places because the J-Machine does not have floating point hardware, and its communication is much faster than that of the Delta Machine.

Chapter 7

Future Work

Two types of related future work are described in this chapter. The first type includes those modifications which are intended to yield performance improvements to the algorithm. The second type includes changes which improve the quality of the output, or broaden the applicability of the methods described in this thesis.

7.1 Performance Improvements

The biggest problem remaining in the algorithm is the superlinear growth of communication costs. This prevents the algorithms from being scalable to arbitrary numbers of processors. The most promising directions for future work are those which directly attack communications costs; several of these are outlined here.

7.1.1 Future Work: the Rotation-Invariant Partition

Image partitioning methods suffer from a significant problem: even under the constraints of viewpoint coherence, there are some changes of viewpoint that could require *all* voxels to move. This would incur an enormous communications expense. For instance, suppose the observer were in the center of a uniformly distributed field of voxels. The proper load balance would be the equally-spaced distribution of processors used as the initial default distribution in frame 0 on both sides of Figure 4.2. Suppose the viewpoint rotated such that each voxel were required to move $\frac{1}{\sqrt{N}}$ the width of the screen, where N is the number of processors. (Recall that, for N processors, the screen is divided into $\sqrt{N}$ parts in each dimension.) This would require that each voxel to move to the adjacent processor. This expense is avoided when the data is concentrated in one area of the field of view, as in Figures 4.4 and 4.4, because the dynamic load balancing algorithm can follow the movement of the data. In this worst-case example, however, the proper load balance both before and after the movement is the same uniform distribution of processors, because the voxels are distributed uniformly.

This looming catastrophe may be dispelled with a new class of partitioning methods, a hybrid of image partitions and object partitions, which we call *rotation-invariant partitions.* We have not yet implemented these methods, but propose them here. These partitions are expected to eliminate communications costs due to viewpoint rotations about the X and Y axes. A more complex variant could also avoid communications arising from rotations of the viewpoint about

the Z axis. The essence of the approach is this: instead of moving the voxels between processors in response to a viewpoint rotation, as required by an image partition, we scroll the processors across the screen.

In the image partition presented in Section 3.2.2, the origin of the coordinate system of the lines dividing the screen, which we call *processor space*, is the bottom left corner of the screen, which is a fixed point in image space. The three axes of processor space are also fixed to those of image space. If, instead, the flat surface of the screen (note that the screen is a bounded plane at a fixed Z value in image space) were mapped onto a torus, and the origin point of processor space were allowed to migrate about on the torus, then the lines dividing the screen could, as a group, be made to follow the movement of the voxels during X or Y viewpoint rotations. If this were done properly, then a pure rotation about the X axis or the Y axis, or both, with no component of translation, need not require any movement of voxels across lines whatsoever. For a leftward viewpoint rotation of 360 degrees, if the screen filled a V degree field of view, then each processor's rendering area would scroll across the screen from left to right $\frac{360}{V}$ times.

Communication due to viewpoint rotations about the Z axis, which are generally less common and less extreme than those about the X or Y axes, could also be eliminated if the axes of processor space were allowed to rotate about the Z axis. This would introduce additional complication, however, because the sides of the rectangular areas rendered by each processor would not remain parallel to the X and Y axes of image space.

Communication incurred by panning (X or Y pure translation) would not be completely eliminated by this technique, but we expect that it would be greatly reduced, especially for voxels that were far from the observer. Communication due to zooming (Z translation) is already low because of how image space is subdivided: processors own everything behind their area of screen space, out to infinity. The lines dividing the screen would, additionally, continue to move relative to one another as required by load balancing; load balancing, by intention, requires that some voxels move to new processors.

Therefore, if both parts of this modification was applied, the only remaining sources of communication should be the movement of load balancing lines; sending voxels to sleep and awakening them as they passed in and out of the field of view; and translation of viewpoint. Again, however, we have not yet implemented and evaluated these partitions.

7.1.2 Giving All Processors More of Load Balancing Tree

This optimization attempts to reduce the number of required hops by letting the processors send voxels directly, or more directly, to their owners.

In the early design stages of this work, a design decision was made to completely distribute the tree of lines dividing the screen: no processor would store line information which was not directly related to its own location on the screen. Another method would be to give all the processors complete information about the entire screen: Giving all processors wider local information would allow them to more accurately estimate local work density, and more importantly, eliminate many hops.

If processors were given the entire load balancing tree, that is, the positions of all lines, then they could send any voxel to its proper owner in a single hop. The total number of hops required would be reduced from:

$$\sum_{L=0\ldots2\ldots(\log N)-1} 2^{\frac{L}{2}} \frac{1}{2}(\log N - L) \qquad \text{for } \log N \text{ odd.}$$

to:

$$\sum_{L=0\ldots2\ldots(\log N)-1} 2^{\frac{L}{2}} \qquad \text{for } \log N \text{ odd.}$$

which is equivalent to:

$$\sqrt{2N} - 1 \qquad \text{for } \log N \text{ odd.}$$

That is, the number of hops would increase as the square root of N, meaning that this part of the communication would scale *better* than linearly — the cost per processor would actually go down as processor number increased.

However, this method would entail other costs: most importantly the cost of broadcasting all the information to all the processors at each frame. The communications per processor for this does increase with increasing number of processors: using a binary tree to implement such a broadcast requires $\log N$ steps which must be done sequentially. Furthermore, the amount of information to represent the entire division of the screen increases linearly with the number of processors each processor would have to store this information.

The choice to implement the former extreme (processors have only local line information) versus the latter extreme (processors all have global line information) was a conscious choice — we were most interested in exploring a completely distributed model. The exploration so far has been informative, and opens the way to another area of exploration in between the two extremes: how much of the tree is the optimal amount to give the processors?

7.1.3 Quadtree and Higher Order Divisions of Screen

Instead of dividing the screen into two at each level, it could be divided into four sections (or more). A load balancing decision would then consist of placing one horizontal and one vertical line (or more) at once. Along with decreasing the number of hops, this would also give the processors making the line positioning decisions a greater area for which they had accurate load information — they would have information for the union of four processors' areas (or more), rather than just two. This would improve load balance at greater rates of data movement.

7.1.4 Measuring the Costs of Load Balancing

Whereas much benefit results from maintaining good dynamic load balance, it does incur some cost: the shifting of the lines partitioning image spacae may incur significant communications expense. The goal of maximizing performance should take into account these relative costs and benefits, perhaps by defining some more general function to optimize.

7.2 Qualitative Improvements

The following possible future additions are not performance improvements but rather add additional functionality to the algorithm.

7.2.1 True Shadowing by Shadow Casting

The implementation as it stands has no provision for shadows. However, the image space subdivision and voxel distribution algorithms presented here could be modified to perform true shading of the data set. In the same way that voxels are now transformed into image space coordinates and composition is performed, the position of a light source could define *light-source space*, and *shadow casting* could be performed. It would not be efficient to do this at each frame, since the data would be thrown repeatedly between the image space and light-source space coordinate systems, but as the data is loaded it would probably be affordable to light it from one or two directions before starting to render it from the observer's viewpoint.

7.2.2 Optimization for Stereo Pairs of Views

The algorithm is now optimized to render from sequences of nearby viewpoints coming from a head mounted display; another source of nearby viewpoints are pairs of stereo views. Some extensions to the idea of master and shadow copies of voxels could be applicable here; for instance, voxels could be labelled *left*, *right*, or *both*. This labelling could eliminate the need for rapid shifting of many voxels in between rendering of left and right stereo views.

7.2.3 Movie Loop

Voxels could contain, rather than a single scalar value, a time sequence or a "movie loop" of scalar values. In this way, an animated data set could be played without the expense of loading new data sets at every frame. The trade-off would be increased cost because of the larger storage requirements of voxels.

Chapter 8

Conclusions

This thesis contributes several new techniques which may be applied to the concurrent the volume rendering of large unstructured scalar data sets.

A dynamic partitioning of the image space is described which arranges the data in the concurrent machine so that, once the data has been distributed, composition may be performed without communication between processors. This dynamic partitioning is particularly efficient at rendering a sequence of nearby viewpoints, such as those arising from continuous head motion: for rendering from a nearby subsequent viewpoint, most data will not need to be redistributed. A tree-based technique to transport the voxels of unstructured data to the correct processors for composition is described. An accurate technique to load balance the concurrent algorithm, by dynamically changing the partitioning, is also detailed.

These ideas have been implemented on the concurrent Intel Delta Machine and applied to the visualization of unstructured scalar volumes originating from MRI data.

The main problem remaining with the current algorithm is the cost of communication to redistribute the voxels as the number of processors increases. A characteristic common to all image partitions — that some viewpoint movements may require communication of all voxels — also remains. Several directions for future work, most notably several which decrease these communication costs dramatically, have been suggested.

Bibliography

[Appel68] A. Appel. "Some techniques for shading machine rendering of solids." *Proc. AFIS, Spring Joint Computer Conference*, 32:37-45. 1968.

[Blinn82] J.F. Blinn. "Light reflection functions for simulation of clouds and dusty surfaces." *ACM Computer Graphics*, 16(bf 3):21-29. 1982.

[Bouknight70] W.K. Bouknight, K.C. Kelley. "An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources." *AFIPS Conf. Proc.* 36:1-10. 1970.

[Phong75] Phong Bui-Tuong. "Ilumination for Computer Generated Pictures". *CACM*, 18(**6**):311-317. June, 1975.

[Cleary86] J.G. Cleary, B.M. Wyvill, G.M. Birtwistle, R. Vatti. "Multiprocessor Ray Tracing." *Computer Graphics Forum*, 5:3-12. North-Holland. 1986.

[Dally89] W.J. Dally, et al., "The J-Machine: A Fine-grain Concurrent Computer," *Information Processing 89*, G. X. Ritter (ed.), Elsevier Science Publishers B.V., North Holland, IFIP, 1989.

[Dippé84] M. Dippé, J. Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis." *ACM Computer Graphics* 18(**3**):149-158, 1984.

[Fujimoto86] A. Fujimoto, T. Tanaka, K. Iwata. "ARTS: Accelerated ray-tracing system." *IEEE Computer Graphics Appl.* 6(**4**):16-26. April, 1986.

[Drebin88] R.A. Drebin, L. Carpenter, P. Hanrahan. "Volume rendering." *ACM Computer Graphics* 22(**4**):65-74, 1988.

[Glassner84] A.S. Glassner. "Space subdivision for fast ray tracing." *IEEE Computer Graphics Appl.* 4(**10**):15-22. October, 1984.

[Goldsmith87] J. Goldsmith, J. Salmon. "Automatic creation of object hierarchies for ray tracing." *IEEE Computer Graphics Appl.*, 7(**5**):14-20. May, 1987.

[Gouraud71] Gouraud, H. "Continuous Shading of Curved Surfaces." *IEEE Trans. on Computers*, C-20(**6**):623-629. June, 1971.

[Green90] S.A. Green, D.J. Paddon. "A highly flexible multiprocessor solution for ray tracing." *The Visual Computer* (1990), 6:62-73. 1990.

[Hall83] R.A. Hall, D.P. Greenberg. "A testbed for realistic image synthesis." *IEEE Comput. Graph. Appl.* 3(**10**):10-20. 1983.

[Kajiya84] J.T. Kajiya, B.P. Von Herzen. "Ray tracing volume densities." *ACM Computer Graphics* 18(**3**):165-174. 1984.

[Kaplan85] M.R. Kaplan. "Space tracing a constant time ray tracer." in *State of the Art in Image Synthesis* (SIGGRAPH85 course notes). July 1985.

[Kay79] D.S. Kay. *Transparency, Refraction, and Ray Tracing for Computer Synthesized Images.* Master's Thesis, Cornell University, Ithaca, N.Y., Jan, 1979.

[Kay86] T.L. Kay, J.T. Kajiya. "Ray tracing complex scenes." *ACM Computer Graphics.* 20(**4**):269-278. 1986.

[Kobayashi87] H. Kobayashi, T. Nakmura, Y. Shigei. "Parallel processing of an object space for image synthesis using ray tracing." *The Visual Computer* (1987), 3:13-22, Springer-Verlag 1987.

[Kobayashi88] H. Kobayashi, S. Nishimura, K. Kubota, T. Nakamura, Y. Shigei. "Load balancing strategies for a parallel ray-tracing system based on constant subdivision". *The Visual Computer* (1988), 4:197-209, Springer-Verlag 1988.

[Levoy88] M. Levoy. "Display of surfaces from volume data." *IEEE Computer Graphics and Applications*, May 1988:29-37.

[Levoy89] M. Levoy. "Design for a real-time high-quality volume rendering workstation." *Chapel Hill Workshiop on Volume Visualization*, May 1989:85-92.

[Lorensen87] W.E. Lorensen, H.E. Cline. "Marching cubes: A high resolution 3d surface construction algorithm." *ACM Computer Graphics*, 21(4):163-169. 1988.

[Laur91] D. Laur, P. Hanrahan. "Hierarchical splatting: A progressive refinement algorithm for volume rendering." *ACM Computer Graphics*, 25(4):285-288. 1991.

[Max86] N.L. Max. "Light diffusion through clouds and haze." *Computer Vision, Graphics, and Image Processing*, 33:280-292. 1986.

[Max90] N. Max, P. Hanrahan, R. Crawfis. "Area and volume coherence for efficient visualization of 3d Scalar functions." *ACM Computer Graphics*, 24(**5**):27-33. 1990.

[Nemoto86] K. Nemoto, T. Omachie. "An adaptive subdivision by sliding boundary surfaces for fast ray tracing." In: *Proc Graphics Interface '86, Canadian Information Processing Society*, pp. 43-48.

[Neumann93] U. Neumann. *Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis.* Ph.D. Thesis, Dept. of Computer Science, University of North Carolina, 1993.

[Nieh92] J. Nieh, M. Levoy. "Volume rendering on scalable shared-memory MIMD architectures." *ACM SIGGRAPH 1992 Workshop on Volume Visualization*, pp. 17-24.

[Nishimura83] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, K. Omura. "LINKS-1, a parallel pipelined multimicrocomputer system for image creation." In: *Proc. 10th Symposium on Computer Architecture, SIGARCH.* 1983.

[Priol89] T. Priol, K. Bouatouch. "Static load balancing for a parallel ray tracing on a MIMD hypercube." *The Visual Computer* (1989), 5:109-119, Springer-Verlag 1989.

[Porter84] T. Porter, T. Duff. "Compositing digital images." *ACM Computer Graphics*, 18(**3**):253-259. 1984.

[Upson88] C. Upson, M. Keeler. "V-BUFFER: Visible volume rendering." *ACM Computer Graphics*, 22(**4**):59-65. 1988.

[Rubin80] S. Rubin, T. Whitted, C. "A three-dimensional representation for fast rendering of complex scenes." *ACM Computer Graphics*, 14(**3**):110-116. 1980.

[Sabella88] P. Sabella. "A rendering algorithm for visualizing 3d scalar fields". *ACM Computer Graphics*, 22(**4**):51-58. 1988.

[Salmon88] J. Salmon, J. Goldsmith. "A hypercube ray-tracer." In: *Proceedings of the 3rd Conference on Hypercube Computers and Applications.*

[Scherson87] I.D. Scherson, E. Caspary. "Data structures and the time complexity of ray tracing." *The Visual Computer* (1987), 3:201-213.

[Scherson88] I.D. Scherson, E. Caspary. "Multiprocessing for ray tracing: a hierarchical self-balancing approach." *The Visual Computer* (1988), 4:188-196.

[Sutherland74] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker. "A characterization of ten hidden-surface algorithms." *Comput. Surv.* 6(**1**), 1-55, March 1974.

[Westover90] L. Westover. "Footprint evaluation for volume rendering." *ACM Computer Graphics*, 24(**4**):367-376. 1990.

[Whitted80] T. Whitted. "An improved illumination model for shaded display." *Commun. ACM*, 23(**6**):343-349. June 1980.

[Wilhelms91a] J. Wilhelms, A. Van Gelder. "A Coherent Projection Approach for Direct Volume Rendering." *ACM Computer Graphics*, 25(**4**):275-284. 1991.

[Wilhelms91b] J. Wilhelms. "Decisions in Volume Rendering". In: *State of the Art in Volume Visualization*, Course notes of SIGGRAPH91, 8:I.1-I.11. 1991.

[Williams90] P. Williams and P. Shirley. "A priori algorithms for polyhedral depth ordering and point location." *Technical Report 1018*, University of Illinois at Urbana, 1990.